

PyTorch

Освещающая глубокое
обучение

Эли Стивенс
Лука Антига
Томас Виман

Предисловие Сумита Чинталы



Deep Learning with PyTorch

ELI STEVENS, LUCA ANTIGA,
AND THOMAS VIEHMANN
FOREWORD BY SOUMITH CHINTALA



MANNING
SHELTER ISLAND

PyTorch

Освещающая глубокое обучение

Эли Стивенс, Лука Антига, Томас Виман
Предисловие Сумита Чинталы



Санкт-Петербург • Москва • Минск

2022

ББК 32.813
УДК 004.8
С80

Стивенс Эли, Антига Лука, Виман Томас

С80 PyTorch. Освещающая глубокое обучение. — СПб.: Питер, 2022. — 576 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1945-5

Многие средства глубокого обучения используют Python, но именно библиотека PyTorch настоящему «питоническая». Легкая в освоении для тех, кто знаком с NumPy и scikit-learn, PyTorch упрощает работу с глубоким обучением, обладая в то же время богатым набором функций. PyTorch прекрасно подходит для быстрого создания моделей и без проблем масштабируется до корпоративного проекта. PyTorch используют такие компании, как Apple и JPMorgan Chase.

Навыки работы с этой библиотекой пригодятся вам для карьерного роста. Вы научитесь создавать нейронные сети и системы глубокого обучения с помощью PyTorch. Книга поможет быстро приступить к созданию реального проекта с нуля. В ней описаны лучшие практики всего конвейера работы с данными, включая PyTorch Tensor API, загрузку данных на Python, мониторинг обучения и визуализацию полученных результатов.

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

ББК 32.813
УДК 004.8

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617295263 англ.
ISBN 978-5-4461-1945-5

© 2020 by Manning Publications Co. All rights reserved
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Библиотека программиста», 2022

Краткое содержание

Предисловие	20
Введение.	22
Благодарности	24
Об этой книге.	26
Об авторах.	33
Об иллюстрации на обложке	34

ЧАСТЬ I **Основы PyTorch**

Глава 1. Знакомство с глубоким обучением и библиотекой PyTorch	36
Глава 2. Предобученные сети.	52
Глава 3. В начале был тензор...	78
Глава 4. Представление реальных данных с помощью тензоров	112

6 Краткое содержание

Глава 5. Внутренняя кухня обучения.149

Глава 6. Аппроксимация данных с помощью нейронной сети191

Глава 7. Различаем птиц и самолеты: обучение на изображениях217

Глава 8. Обобщение с помощью сверток249

ЧАСТЬ II

Обучение на изображениях на практике: раннее выявление рака легких

Глава 9. Применение PyTorch в борьбе с раком296

Глава 10. Объединение источников данных.319

Глава 11. Обучение модели классификации обнаружению
потенциальных опухолей348

Глава 12. Улучшение процесса обучения с помощью метрик
и дополнений391

Глава 13. Поиск потенциальных узелков с помощью сегментации435

Глава 14. Сквозной анализ узелков и дальнейшее развитие проекта488

ЧАСТЬ III

Развертывание

Глава 15. Развертывание в производстве534

Оглавление

Предисловие	20
Введение	22
Благодарности	24
Об этой книге	26
Для кого предназначена эта книга	26
Структура издания	27
О коде	29
Прочие источники информации в интернете	31
От издательства	32
Об авторах	33
Об иллюстрации на обложке	34

ЧАСТЬ I **Основы PyTorch**

Глава 1. Знакомство с глубоким обучением и библиотекой PyTorch	36
1.1. Революция глубокого обучения	37
1.2. Использование PyTorch для глубокого обучения	39

8 Оглавление

1.3. Почему PyTorch?	41
1.3.1. Общая картина сферы глубокого обучения	42
1.4. Обзор средств поддержки библиотекой PyTorch проектов глубокого обучения	44
1.5. Аппаратные и программные требования	48
1.5.1. Блокноты Jupyter	49
1.6. Упражнения	50
1.7. Резюме	51
Глава 2. Предобученные сети.	52
2.1. Предобученные сети для распознавания тематики изображения	53
2.1.1. Получение предобученной сети для распознавания изображений	56
2.1.2. AlexNet	57
2.1.3. ResNet	59
2.1.4. На старт, внимание, почти что марш	59
2.1.5. Марш!	62
2.2. Предобученная модель, создающая все лучшие подделки	64
2.2.1. Игра GAN.	65
2.2.2. CycleGAN.	67
2.2.3. Сеть, превращающая лошадей в зебр	68
2.3. Предобученная сеть для описания обстановки	72
2.3.1. NeuralTalk2.	73
2.4. Torch Hub.	74
2.5. Итоги главы	76
2.6. Упражнения	76
2.7. Резюме	77
Глава 3. В начале был тензор...	78
3.1. Мир как числа с плавающей запятой	79
3.2. Тензоры: многомерные массивы	81
3.2.1. От списков Python к тензорам PyTorch	81
3.2.2. Создаем наши первые тензоры	82
3.2.3. Что такое тензоры	83

3.3. Доступ к тензорам по индексам	86
3.4. Поименованные тензоры	86
3.5. Типы элементов тензоров.	90
3.5.1. Задание числового типа с помощью dtype	91
3.5.2. dtype на все случаи жизни	91
3.5.3. Работа с атрибутом dtype тензоров.	92
3.6. API тензоров.	93
3.7. Тензоры: хранение в памяти	95
3.7.1. Доступ к хранилищу по индексу	95
3.7.2. Модификация хранимых значений: операции с заменой на месте.	97
3.8. Метаданные тензоров: размер, сдвиг и шаг.	97
3.8.1. Представления хранилища другого тензора	97
3.8.2. Транспонирование без копирования	100
3.8.3. Транспонирование при более высокой размерности	101
3.8.4. Непрерывные тензоры	102
3.9. Перенос тензоров на GPU	104
3.9.1. Работа с атрибутом device тензоров	105
3.10. Совместимость с NumPy	106
3.11. Обобщенные тензоры тоже тензоры	107
3.12. Сериализация тензоров	108
3.12.1. Сериализация в HDF5 с помощью h5py.	109
3.13. Итоги главы	110
3.14. Упражнения	111
3.15. Резюме.	111
Глава 4. Представление реальных данных с помощью тензоров	112
4.1. Работа с изображениями	113
4.1.1. Добавление цветовых каналов.	114
4.1.2. Загрузка файла изображения	115
4.1.3. Изменение схемы расположения	115
4.1.4. Нормализация данных	117
4.2. Трехмерные изображения: объемные пространственные данные	118
4.2.1. Загрузка данных в специализированном формате	119

4.3. Представление табличных данных	120
4.3.1. Реальный набор данных.	120
4.3.2. Загрузка тензора данных по вину.	122
4.3.3. Представление оценок.	124
4.3.4. Быстрое кодирование	125
4.3.5. Когда считать данные категориальными	127
4.3.6. Поиск пороговых значений	128
4.4. Временные ряды	131
4.4.1. Добавляем измерение времени	132
4.4.2. Компоновка данных по периоду времени.	133
4.4.3. Готов для обучения.	135
4.5. Представление текста	138
4.5.1. Преобразование текста в числа	139
4.5.2. One-hot-кодирование символов.	139
4.5.3. Унитарное кодирование целых слов	141
4.5.4. Вложения текста	144
4.5.5. Вложения текста как схема.	146
4.6. Итоги главы	147
4.7. Упражнения	147
4.8. Резюме	148
Глава 5. Внутренняя кухня обучения.	149
5.1. Всегда актуальный урок моделирования	150
5.2. Обучение — это просто оценка параметров	152
5.2.1. «Жаркая» задача	154
5.2.2. Сбор данных	154
5.2.3. Визуализация данных	154
5.2.4. Выбираем линейную модель для первой попытки	155
5.3. Наша цель — минимизация потерь.	156
5.3.1. Возвращаемся от задачи к PyTorch.	157
5.4. Вниз по градиенту	160
5.4.1. Снижение потерь	161
5.4.2. Выражаем аналитически	162
5.4.3. Подгонка модели в цикле.	164

5.4.4. Нормализация входных сигналов	167
5.4.5. Визуализируем (снова)	170
5.5. Компонент autograd PyTorch: обратное распространение всего чего угодно	171
5.5.1. Автоматическое вычисление градиента	171
5.5.2. Оптимизаторы на выбор	175
5.5.3. Обучение, проверка и переобучение	180
5.5.4. Нюансы автоматического вычисления градиентов и его отключение.	186
5.6. Итоги главы	189
5.7. Упражнение	189
5.8. Резюме	190
Глава 6. Аппроксимация данных с помощью нейронной сети	191
6.1. Искусственные нейроны	192
6.1.1. Формирование многослойной сети.	194
6.1.2. Функция ошибки.	195
6.1.3. Все, что нам нужно, — это функция активации	195
6.1.4. Другие функции активации	198
6.1.5. Выбор наилучшей функции активации	199
6.1.6. Что обучение дает нейронной сети	200
6.2. Модуль nn PyTorch.	203
6.2.1. Использование метода <code>__call__</code> вместо метода <code>forward</code>	204
6.2.2. Обратно к линейной модели	205
6.3. Наконец-то нейронная сеть	210
6.3.1. Замена линейной модели	210
6.3.2. Просматриваем информацию о параметрах	212
6.3.3. Сравнение с линейной моделью.	214
6.4. Итоги главы	215
6.5. Упражнения	215
6.6. Резюме	216
Глава 7. Различаем птиц и самолеты: обучение на изображениях	217
7.1. Набор крошечных изображений	218
7.1.1. Скачиваем CIFAR-10	218
7.1.2. Класс Dataset	219

7.1.3. Преобразования объектов Dataset	221
7.1.4. Нормализация данных	223
7.2. Различаем птиц и самолеты	226
7.2.1. Формирование набора данных	227
7.2.2. Полносвязная модель	227
7.2.3. Выходной сигнал классификатора	229
7.2.4. Представление выходного сигнала в качестве вероятностей	230
7.2.5. Функция потерь для классификации	234
7.2.6. Обучение классификатора	237
7.2.7. Ограничения, накладываемые полносвязностью	244
7.3. Итоги главы	246
7.4. Упражнения	247
7.5. Резюме	248
Глава 8. Обобщение с помощью сверток	249
8.1. Аргументы в пользу сверток	250
8.1.1. Что делают свертки	250
8.2. Свертки в действии	253
8.2.1. Дополнение нулями по краям	255
8.2.2. Обнаружение признаков с помощью сверток	257
8.2.3. Расширяем кругозор с помощью субдискретизации и повышения глубины сети	260
8.2.4. Собираем нашу нейронную сеть воедино	264
8.3. Создание подклассов nn.Module	266
8.3.1. Наша сеть как подкласс nn.Module	267
8.3.2. Как PyTorch отслеживает параметры и подмодули.	269
8.3.3. Функциональные API	270
8.4. Обучаем нашу сверточную сеть	272
8.4.1. Измерение степени безошибочности.	274
8.4.2. Сохранение и загрузка модели	275
8.4.3. Обучение на GPU	275
8.5. Архитектура модели	277
8.5.1. Расширение объема памяти: ширина	278

8.5.2. Улучшаем сходимость модели и ее способности к обобщению: регуляризация	280
8.5.3. Забираемся глубже для усвоения более сложных структур: глубина сети	285
8.5.4. Сравнение архитектур этого раздела.	291
8.5.5. Описанное здесь уже устарело	292
8.6. Итоги главы	292
8.7. Упражнения	293
8.8. Резюме	294

ЧАСТЬ II

Обучение на изображениях на практике: раннее выявление рака легких

Глава 9. Применение PyTorch в борьбе с раком	296
9.1. Постановка задачи	296
9.2. Подготовка к масштабному проекту	298
9.3. Что такое компьютерная томография	300
9.4. Проект: сквозной детектор рака легких	303
9.4.1. Почему нельзя просто передавать данные в нейронную сеть, пока она не заработает	308
9.4.2. Что такое узелок	313
9.4.3. Наш источник данных: The LUNA Grand Challenge	315
9.4.4. Загрузка данных LUNA	315
9.5. Итоги главы	317
9.6. Резюме	317
Глава 10. Объединение источников данных.	319
10.1. Файлы необработанных данных КТ	321
10.2. Парсинг данных аннотаций LUNA	322
10.2.1. Обучающие и проверочные наборы.	324
10.2.2. Объединение аннотаций и данных кандидатов	324
10.3. Загрузка сканов КТ.	327
10.3.1. Единицы Хаунсфилда	330
10.4. Определение положения узелка в системе координат пациента.	331
10.4.1. Система координат пациента.	332

10.4.2. Форма КТ-скана и размеры вокселя	334
10.4.3. Преобразование миллиметров в адреса вокселей	336
10.4.4. Извлечение узелка из скана КТ	337
10.5. Простая реализация Dataset	339
10.5.1. Кэширование массивов-кандидатов с помощью функции getCtRawCandidate	342
10.5.2. Построение набора данных в LunaDataset.__init__.	343
10.5.3. Разделение данных на обучающие и проверочные	343
10.5.4. Отображение данных.	345
10.6. Итоги главы	346
10.7. Упражнения	346
10.8. Резюме.	347
Глава 11. Обучение модели классификации обнаружению	
потенциальных опухолей	348
11.1. Базовая модель и цикл обучения	348
11.2. Точка входа приложения	352
11.3. Предварительная настройка и инициализация	354
11.3.1. Инициализация модели и оптимизатора	355
11.3.2. Передача данных загрузчикам	356
11.4. Первый сквозной дизайн нейронной сети.	359
11.4.1. Основы свертки	360
11.4.2. Полная модель.	363
11.5. Обучение и проверка модели.	366
11.5.1. Функция calculateBatchLoss	368
11.5.2. Цикл проверки работает аналогично.	370
11.6. Вывод метрик производительности	371
11.6.1. Функция logMetrics	372
11.7. Запуск обучения	375
11.7.1. Необходимые для обучения данные	377
11.7.2. Интерлюдия: функция enumerateWithEstimate	378
11.8. Оценка модели: 99,7 % правильных ответов — это отличный результат, не так ли?	379
11.9. Построение графиков для метрик обучения с помощью TensorBoard	381

11.9.1. Запуск TensorBoard.	381
11.9.2. Внедрение TensorBoard в функцию регистрации метрик . . .	385
11.10. Почему модель не учится обнаруживать узелки?	387
11.11. Итоги главы.	388
11.12. Упражнения.	389
11.13. Резюме.	389
Глава 12. Улучшение процесса обучения с помощью метрик и дополнений.	391
12.1. План модернизации	392
12.2. Хорошие собаки против плохих парней: ложноположительные и ложноотрицательные результаты	393
12.3. Визуализация положительных и отрицательных результатов . . .	395
12.3.1. Высокий отклик Рокси	398
12.3.2. Высокая точность Престона	400
12.3.3. Реализация точности и отклика в logMetrics	401
12.3.4. Готовая метрика производительности: метрика F1	402
12.3.5. Как модель работает с новыми метриками	407
12.4. Как выглядит идеальный набор данных.	408
12.4.1. Как сделать данные более «идеальными»	411
12.4.2. Сравнение результатов обучения по сбалансированному и несбалансированному набору	417
12.4.3. Распознавание симптомов переобучения	419
12.5. Вернемся к проблеме переобучения	421
12.5.1. Модель прогнозирования с переобучением по возрасту. . . .	421
12.6. Предотвращение переобучения путем увеличения набора данных	422
12.6.1. Методы дополнения данных	423
12.6.2. Наблюдение за улучшением данных после дополнения. . . .	429
12.7. Итоги главы	431
12.8. Упражнения	432
12.9. Резюме.	433
Глава 13. Поиск потенциальных узелков с помощью сегментации	435
13.1. Добавим в проект вторую модель	436
13.2. Различные типы сегментации	438

13.3. Семантическая сегментация: попиксельная классификация	439
13.3.1. Архитектура U-Net	443
13.4. Обновление модели сегментации	445
13.4.1. Адаптация готовой модели к нашему проекту	447
13.5. Модификация набора данных для сегментации	449
13.5.1. Особые требования U-Net к размеру входных данных.	450
13.5.2. Компромиссы U-Net при работе с 3D- и 2D-данными	450
13.5.3. Формирование достоверных данных.	452
13.5.4. Реализация Luna2dSegmentationDataset	459
13.5.5. Разработка наших данных для обучения и проверки.	464
13.5.6. Реализация набора данных TrainingLuna2dSegmentation	465
13.5.7. Дополнение данных на ГП	466
13.6. Внедрение сегментации в сценарий обучения	469
13.6.1. Инициализация наших моделей сегментации и увеличения	470
13.6.2. Использование оптимизатора Adam	471
13.6.3. Потеря Дайса.	471
13.6.4. Получение изображений в TensorBoard	475
13.6.5. Обновление логирования метрик	479
13.6.6. Сохранение модели.	481
13.7. Результаты	482
13.8. Итоги главы	485
13.9. Упражнения	486
13.10. Резюме	487
Глава 14. Сквозной анализ узелков и дальнейшее развитие проекта	488
14.1. Финишная прямая	488
14.2. Независимость проверочного набора	492
14.3. Объединение сегментации КТ и классификации узелков-кандидатов.	493
14.3.1. Сегментация	494
14.3.2. Группировка вокселей в узелки-кандидаты.	495
14.3.3. Узелок или не узелок? Классификация и снижение числа ложноположительных результатов	497
14.4. Количественная оценка	502

14.5. Прогнозирование злокачественности	503
14.5.1. Получение информации о злокачественных новообразованиях	503
14.5.2. Базовый уровень для вычисления площади под кривой: классификация по диаметру	504
14.5.3. Повторное использование весов: тонкая настройка.	508
14.5.4. Больше данных в TensorBoard	515
14.6. Каков диагноз?	518
14.6.1. Наборы для обучения, проверки и тестирования	520
14.7. Что дальше? Дополнительные источники вдохновения (и данных).	521
14.7.1. Борьба с переобучением: выбор лучшей регуляризации.	522
14.7.2. Подготовка обучающих данных	525
14.7.3. Итоги конкурса и научные работы	527
14.8. Итоги главы	528
14.8.1. За кулисами	529
14.9. Упражнения	530
14.10. Резюме	531

ЧАСТЬ III

Развертывание

Глава 15. Развертывание в производстве	534
15.1. Поставка моделей PyTorch	535
15.1.1. Размещение модели на сервере Flask.	536
15.1.2. Требования к развертыванию	538
15.1.3. Пакетная обработка запросов	539
15.2. Экспорт модели	545
15.2.1. Совместимость за пределами PyTorch с ONNX	546
15.2.2. Встроенный механизм экспорта PyTorch: отслеживание	547
15.2.3. Сервер с отслеженной моделью	549
15.3. Взаимодействие с PyTorch JIT.	549
15.3.1. Что за пределами Python/PyTorch	550
15.3.2. Двойственная природа PyTorch как интерфейса и бекэнда	552
15.3.3. TorchScript	552

15.3.4. Использование сценариев как лучшей замены отслеживания.556
15.4. LibTorch: PyTorch в C++.557
15.4.1. Запуск JIT-моделей из C++558
15.4.2. Сразу работаем с C++ и API C++561
15.5. Добавим мобильности565
15.5.1. Повышение эффективности: проектирование моделей и квантование.569
15.6. Новые технологии: корпоративная поставка моделей PyTorch571
15.7. Итоги главы571
15.8. Упражнение572
15.9. Резюме.572

*Моей жене (эта книга не появилась бы без ее бесценной поддержки и участия),
моим родителям (без них на свет не появился бы я) и моим детям
(если бы не они, эта книга появилась бы намного раньше).*

*Спасибо вам за то, что вы — моя крепость, мой фундамент и моя радость.
Эли Стивенс (Eli Stevens)*

*Аналогично :) Впрочем, на самом деле посвящаю ее вам, Элис и Луиджи.
Лука Антига (Luca Antiga)*

*Еве, Ребекке, Джонатану и Давиду.
Томас Виман (Thomas Viehmann)*

Предисловие

На момент начала работы над проектом PyTorch в середине 2016 года мы были просто группой фанатов программного обеспечения с открытым исходным кодом, которые встретились в интернете и захотели написать лучшее программное обеспечение для глубокого обучения. Двое из трех авторов этой книги, Лука Антига и Томас Виман, сыграли важную роль в разработке фреймворка PyTorch и достижении им того успеха, который он имеет сегодня.

Наша цель с PyTorch заключалась в создании как можно более гибкого фреймворка для реализации алгоритмов глубокого обучения. Мы работали очень сосредоточенно и за относительно короткий промежуток времени подготовили безупречный продукт. Это было бы невозможно, если бы мы не стояли на «плечах» гигантов. PyTorch унаследовал немалую долю своей базы кода из проекта Torch7, начатого в 2007 году Ронаном Колобером (Ronan Collobert) и другими, — истоки этого проекта лежат в языке программирования Lush, начало которому положили Ян Ле Кун (Yann Le Cun) и Леон Ботту (Leon Bottou). Эта богатая история помогла нам сосредоточить свое внимание на необходимых изменениях, вместо того чтобы начинать проектирование с нуля.

Успех PyTorch трудно объяснить каким-то одним фактором. Этот проект обладает удобным интерфейсом и улучшенными возможностями для отладки, что в конечном итоге обеспечивает более высокую продуктивность у пользователей. Широкое распространение PyTorch привело к созданию чудесной экосистемы основанного на нем программного обеспечения и исследований, которая еще больше обогатила этот фреймворк.

Для упрощения изучения PyTorch существует несколько курсов и университетских учебных программ, а также множество онлайн-блогов и руководств. Однако

книг ему посвящено очень мало. В 2017 году, когда кто-то спросил меня: «Когда появится книга по PyTorch?» — я ответил: «Если кто-то сейчас такую и пишет, можете быть уверены, что к моменту выхода в свет она уже будет устаревшей».

С книгой *«PyTorch. Освещающая глубокое обучение»* наконец-то появился полноценный учебный курс по PyTorch. В нем очень подробно рассматриваются базовые понятия и абстракции, разбираются основы структур данных, таких как тензоры и нейронные сети, и дается понимание всех нюансов реализации. Кроме того, здесь охвачены такие продвинутое темы, как JIT (англ. Just-in-Time, компиляция «точно в нужное время») и развертывание для промышленной эксплуатации (один из аспектов PyTorch, который в настоящее время не освещается ни в одной другой книге).

Кроме того, в книге описаны прикладные задачи: вы пройдете через все этапы использования нейронных сетей для решения сложной и важной медицинской проблемы. Благодаря глубоким знаниям Луки в сфере биоинженерии и медицинской визуализации, опыту Эли по созданию на практике программного обеспечения для медицинских устройств и выявления заболеваний, а также багажу знаний Томаса как разработчика ядра PyTorch этот путь будет пройден так основательно, как и должно быть.

В общем и целом, я надеюсь, что эта книга станет для вас «расширенным» справочником и важной составной частью вашей библиотеки.

*Сумит Чинтала (Soumith Chintala),
один из создателей PyTorch*

Введение

В 1980-х годах, еще детьми, делая первые шаги на наших Commodore VIC 20 (Эли), Sinclair Spectrum 48K (Лука) и Commodore C16 (Томас), мы наблюдали расцвет персональных компьютеров, учились программировать и писать алгоритмы на все более быстрых машинах и часто мечтали о том, куда приведут нас компьютеры. Мы остро ощущали пропасть между тем, что делали компьютеры в фильмах, и тем, на что они были способны в реальности, дружно закатывая глаза, когда главный герой в шпионском боевике говорил: «Компьютер, повысь качество изображения».

Позднее, уже во время профессиональной деятельности, двое из нас, Эли и Лука, независимо друг от друга занялись анализом медицинских данных и столкнулись с одинаковыми проблемами при написании алгоритмов, способных справиться с естественным разнообразием параметров человеческого тела. Существует множество эвристических правил выбора наилучшего сочетания алгоритмов, позволяющих добиться нужных результатов и спасти положение. Томас на рубеже столетий изучал нейронные сети и распознавание закономерностей, но затем получил докторскую степень по математике, занимаясь моделированием.

В начале 2010-х появилось глубокое обучение (ГО), сначала в сфере машинного зрения, а затем и применительно к задачам анализа медицинских снимков, например к распознаванию различных структур и изменений. Как раз в это время, в первой половине десятилетия, глубокое обучение заинтересовало и нас. Понадобилось немного времени, чтобы осознать, что глубокое обучение представляет собой совершенно новый способ написания программного обеспечения: новый класс универсальных алгоритмов, способных обучаться решению сложных задач на основе наблюдения данных.

Для нас, поколения 80-х, горизонты возможного для компьютеров расширились в один миг. Теперь их ограничивали не умы лучших программистов, а лишь данные, архитектуры нейронных сетей и процесс обучения. Оставалось только заняться ими. Лука выбрал для этой цели Torch 7 (<http://torch.ch/>), уважаемого предшественника PyTorch, гибкого, облегченного и быстрого, с удобочитаемым исходным кодом, написанным на Lua и чистом C, дружелюбным сообществом пользователей и длинной предысторией. Лука влюбился в него с первого взгляда. Единственный серьезный недостаток Torch 7 заключался в его отрыве от непрерывно растущей экосистемы исследования данных Python data science, которой могли пользоваться другие платформы. Эли интересовался ИИ еще в колледже¹, но его карьера пошла в другом направлении, да и прочие, более ранние фреймворки глубокого обучения показались ему слишком неудобными, чтобы с удовольствием использовать их в любительских проектах.

Поэтому мы все очень обрадовались, когда 18 января 2017 года был опубликован первый релиз PyTorch. Лука начал работать над его ядром, а Эли очень быстро присоединился к сообществу пользователей, предлагая множество исправлений ошибок, новые функции и обновления документации. Томас внес тонну функций и исправлений ошибок в PyTorch и в итоге стал полноценным разработчиком ядра. Было ощущение, что начинается нечто громадное, как раз на нужном уровне сложности и с минимальным количеством избыточных умственных усилий. Уроки бережливого проектирования, полученные из Torch 7, были заимствованы, но на этот раз уже с набором современных возможностей, таких как автоматическое дифференцирование, динамические графы вычислений и интеграция NumPy.

С учетом нашей вовлеченности и энтузиазма и после организации нескольких семинаров по PyTorch написание книги представлялось естественным следующим шагом. Мы ставили перед собой задачу написать такую книгу, которая могла бы помочь нам в то время, когда мы только начинали.

Ничего удивительного, что мы начали с грандиозных задач: обучить всем основам, пройти вместе с читателями по проектам от начала до конца и показать наиболее современные и удачные модели на PyTorch. Вскоре мы поняли, что для этого понадобится намного больше одной книги, так что мы решили сосредоточить внимание на изначальной задаче: посвятить время и силы тому, чтобы охватить ключевые идеи PyTorch, не требуя от читателя практически никаких предварительных знаний глубокого обучения, и дойти до уровня, на котором можно будет показать читателям полноценный комплексный проект, для которого мы вернулись к истокам и решили продемонстрировать сложную задачу анализа медицинских снимков.

¹ В те времена, когда «глубокой» нейронная сеть считалась при наличии трех скрытых слоев!

Благодарности

Мы глубоко признательны команде PyTorch. Благодаря их общим усилиям PyTorch органично вырос из проекта уровня летней стажировки в первоклассный инструмент глубокого обучения. Мы хотели бы упомянуть Сумита Чинталу (Soumith Chintala) и Адама Пашке (Adam Paszke), которые, не говоря об их великолепном коде, активно продвигали подход «главное — сообщество пользователей» в управлении данным проектом. Нынешний уровень процветания и терпимости в сообществе PyTorch — явное свидетельство их усилий.

Что касается сообщества, PyTorch не был бы таким, если бы не неустанный труд отдельных пользователей, помогающих как начинающим, так и опытным пользователям на дискуссионном форуме. Среди всех досточтимых участников проекта следует особо отметить нашей признательностью Петра Бялецкого (Pitr Bialecki). А что касается нашей книги, особая благодарность Джо Списаку (Joe Spisak) за веру в ценность ее для сообщества, а также Джефу Смиты (Jeff Smith), выполнившему колоссальный объем работы для того, чтобы эту ценность воплотить в жизнь. Также мы очень благодарны Брюсу Лину (Bruce Lin), который помог нам отдельно подготовить часть I этого текста и обеспечить свободный доступ к ней для сообщества пользователей PyTorch.

Мы хотели бы поблагодарить команду издательства Manning, которая чутко вела нас по этому пути, всегда учитывая хрупкое равновесие между семьей, работой и написанием книги. Спасибо Эрин Туи (Erin Twohey), обратившейся к нам и спросившей, не интересует ли нас написание книги, и Майклу Стивенсу (Michael Stephens), склонившему нас к положительному ответу. Мы ведь *сказали* тебе, что у нас нет времени! Брайан Хэнафи (Brian Hanafee) делал намного больше, чем обязан был сделать рецензент. Артур Зубарев (Arthur Zubarev) и Костас Пассадис (Kostas Passadis) присылали ценные замечания, а Дженнифер Хоул

(Jennifer Houle) сумела справиться с нашим причудливым художественным стилем. Наш выпускающий редактор Тиффани Тейлор (Tiffany Taylor) была очень внимательна к деталям, так что во всех ошибках виноваты мы и только мы. Также мы хотели бы поблагодарить нашего редактора-координатора, Дейрдре Хайэм (Deirdre Hiam), нашего корректора Кэти Теннант (Katie Tennant) и нашего редактора-рецензента Ивана Мартиновича (Ivan Martinovic). Немало людей, мелькавших только в списке скрытых копий веток обсуждения состояния книги, также работало «за кулисами», чтобы довести эту книгу до печати. Спасибо всем, кого мы еще не упомянули! Сделать эту книгу лучше помогли также анонимные рецензенты, с их непредвзятыми отзывами и замечаниями.

Наш неустанный редактор Фрэнсис Левковиц (Frances Lefkowitz) заслуживает медали и недели на тропическом острове за то, что дотянула эту книгу до финишной прямой. Спасибо тебе за всю работу и за изящество, с которым она была проделана.

Мы хотели бы также поблагодарить рецензентов, которые помогли во многом улучшить эту книгу: Александра Ерофеева (Aleksandr Erofeev), Одри Карстенсен (Audrey Carstensen), Башира Чихани (Bachir Chihani), Карлоса Андреса Маришала (Carlos Andres Mariscal), Дэйла Нила (Dale Neal), Дэниела Береша (Daniel Berecz), Донира Улмасова (Doniyor Ulmasov), Эзру Стивенса (Ezra Stevens), Готфрида Асамоа (Godfred Asamoah), Хелен Мэри Лабао Баррамеду (Helen Mary Labao Barrameda), Хильду ван Гисель (Hilde Van Gysel), Джейсона Леонарда (Jason Leonard), Джефа Когшалла (Jeff Cogshall), Костаса Пассадиса (Kostas Passadis), Линси Нил (Linnsey Nil), Мэтью Чжана (Mathieu Zhang), Майкла Константа (Michael Constant), Мигеля Монталво (Miguel Montalvo), Орландо Алехо Мендеса Моралеса (Orlando Alejo Méndez Morales), Филиппе ван Бергена (Philippe Van Bergen), Риз Стивенс (Reece Stevens), Шриниваса К. Рамана (Srinivas K. Raman) и Ючжана Шрестху (Yujan Shrestha).

Всем нашим друзьям и родственникам, недоумевавшим, что мы скрываем эти два года: «Привет! Мы соскучились по вам! Давайте как-нибудь ходим в ресторан».

Об этой книге

Цель книги — изложить основы глубокого обучения с помощью PyTorch и продемонстрировать их в действии на реальном примере. Мы постарались охватить ключевые идеи глубокого обучения и показать, как PyTorch помогает применять их на практике. В этой книге мы сделали все, чтобы вдохновить вас на дальнейшие исследования, для чего избирательно углублялись в подробности, объясняя, что происходит «за кулисами».

«PyTorch. Освещающая глубокое обучение» не стремится к званию справочного руководства, скорее, это схематичный путеводитель, открывающий дорогу к самостоятельному изучению более продвинутых материалов в интернете. Поэтому мы сосредоточились на определенном подмножестве возможностей PyTorch. Наиболее заметно отсутствие упоминания рекуррентных нейронных сетей, но то же справедливо и относительно некоторых других частей API PyTorch.

ДЛЯ КОГО ПРЕДНАЗНАЧЕНА ЭТА КНИГА

Книга предназначена для программистов, стремящихся либо заниматься глубоким обучением, либо просто познакомиться с PyTorch. В роли среднестатистического читателя мы представляем себе специалиста по вычислительной технике, исследователя данных, разработчика программного обеспечения или студента/аспиранта вуза соответствующей специализации. Поскольку мы не требуем предварительных знаний глубокого обучения, некоторые разделы в первой части книги представляют собой повторение понятий, уже хорошо знакомых опытным специалистам. Надеемся, что даже такие читатели смогут взглянуть на уже знакомые вопросы немного под другим углом.

Мы предполагаем, что читатели знакомы с основами императивного и объектно-ориентированного программирования. Поскольку в книге используется язык Python, вы должны быть знакомы с его синтаксисом и рабочей средой. Необходимое условие: уметь устанавливать пакеты и выполнять сценарии Python на вашей платформе. У читателей, работавших ранее на C++, Java, JavaScript, Ruby и других подобных языках программирования, не должно возникнуть проблем с этим, хотя может потребоваться изучить некоторые дополнительные материалы. Аналогично не помешает (а возможно, и необходимо) знакомство с библиотекой NumPy. Мы также ожидаем от вас понимания основ линейной алгебры, в частности знания, что такое матрицы, векторы и скалярное произведение.

СТРУКТУРА ИЗДАНИЯ

Книга разбита на три отдельные части. Часть I охватывает основы, а часть II посвящена комплексному проекту, основанному на описанных в части I базовых понятиях с добавлением более продвинутых. Довольно короткая часть III завершает книгу обзором возможностей PyTorch по развертыванию. Вероятно, вы обратите внимание на различные стили написания и визуализации в разных частях. Хотя эта книга и является результатом бесчисленных часов совместного планирования, обсуждения и редактуры, написание и визуализация в разных частях были распределены между авторами: Лука в основном отвечал за часть I, а Эли — за часть II¹. Томас же старался гармонично объединить свой стиль в части III и отдельных разделах со стилями частей I и II. Вместо того чтобы искать наименьший общий знаменатель, мы решили сохранить характерные для каждой из частей исходные стили авторов.

Ниже приведено разделение каждой части на главы и краткое описание каждой из них.

ЧАСТЬ I

В части I мы начнем знакомство с PyTorch, нарабатывая основные навыки, необходимые для понимания уже существующих проектов PyTorch, а также создания своих собственных. В ней охватываются API PyTorch и некоторые «закулисные» возможности, делающие библиотеку PyTorch такой, какая она есть. Также мы поработаем в ней над обучением нашей первой базовой модели классификации. К концу части I вы будете готовы приступить к настоящему проекту.

Глава 1 познакомит вас с библиотекой PyTorch и ее местом в революции глубокого обучения, а также расскажет о том, что отличает PyTorch от прочих фреймворков глубокого обучения.

¹ В других частях встречается мешанина стилей Эли и Томаса; не удивляйтесь, если стиль будет меняться прямо посередине главы!

Глава 2 показывает PyTorch в действии на примере предобученных сетей, а также объясняет, как скачивать и запускать модели в PyTorch Hub.

В главе 3 вы познакомитесь с основным «кирпичиком» PyTorch — тензором — и его API, а также некоторыми «закулисными» нюансами его реализации.

Глава 4 покажет представление различных типов данных в виде тензоров, а также расскажет, какие формы должны быть у тензоров для моделей глубокого обучения.

Глава 5 описывает техническую сторону обучения с помощью градиентного спуска и то, как PyTorch делает это через автоматическое дифференцирование.

Глава 6 демонстрирует процесс создания и обучения в PyTorch нейронной сети для регрессии с помощью модулей `nn` и `optim`.

Глава 7 продолжает начатое в предыдущей главе и описывает создание полно-связной модели для классификации изображений, а также расширяет ваши знания API PyTorch.

Глава 8 познакомит вас со сверточными нейронными сетями и затронет более сложные вопросы создания нейросетевых моделей и их реализации в PyTorch.

ЧАСТЬ II

Каждая глава в части II будет приближать нас к комплексному решению задачи автоматического обнаружения рака легких. Мы воспользуемся этой непростой задачей для демонстрации реальных подходов к решению таких масштабных проблем, как скрининг рака. Это большой проект с упором на тщательное проектирование, диагностику и устранение неполадок, а также решение проблем.

В главе 9 описывается комплексная стратегия, используемая нами для классификации опухолей легких, начиная со снимков компьютерной томографии (КТ).

В главе 10 мы загрузим данные сформированных людьми описаний вместе с КТ-снимками и преобразуем соответствующую информацию в тензоры с помощью стандартных API PyTorch.

В главе 11 представлена первая модель классификации, работающая на основе описанных в главе 10 обучающих данных. Мы обучим эту модель и соберем простейшие метрики ее работы. Также мы расскажем об использовании TensorBoard для мониторинга обучения.

В главе 12 мы исследуем и внедрим стандартные метрики производительности модели и воспользуемся ими для выявления слабых мест в выполненном ранее обучении. А затем устраним эти изъяны с помощью усовершенствованного

обучающего набора данных с использованием балансировки и дополнения данных.

В главе 13 описывается сегментация, попиксельная архитектура модели, с помощью которой мы сгенерируем карту интенсивности возможных расположений узелков, охватывающую весь КТ-снимок. При помощи этой карты интенсивности можно находить узелки на КТ-снимках, для которых отсутствуют данные сформированных людьми описаний.

В главе 14 реализован окончательный полнофункциональный проект диагностики онкопациентов с помощью нашей новой модели сегментирования с последующей классификацией.

ЧАСТЬ III

Часть III состоит из одной главы, посвященной развертыванию. В главе 15 приведен обзор способов развертывания моделей PyTorch в простом веб-сервисе, встраивания их в программы на C++ или развертывания на мобильных устройствах.

О коде

Весь код в этой книге был написан в расчете на Python 3.6 или более позднюю версию. Исходный код доступен для скачивания с сайта издательства Manning (<https://www.manning.com/books/deep-learning-with-pytorch>) и на GitHub (<https://github.com/deep-learning-with-pytorch/dlwpt-code>). На момент написания книги текущей была версия 3.6.8, именно на ней мы и проверяли примеры из этой книги. Например:

```
$ python
Python 3.6.8 (default, Jan 14 2019, 11:02:34)
[GCC 8.0.1 20180414 on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Командные строки необходимо вводить в ответ на приглашение командной строки Bash, начинающееся с \$ (например, как в строке `$ python` в этом примере). Код внутри обычного текста выглядит вот так.

Начинающиеся с `>>>` блоки кода воспроизводят сеансы интерактивной командной строки Python. Эти символы `>>>` не являются входными данными; строки текста, не начинающиеся с `>>>` или `...`, являются выводимыми результатами работы. В некоторых случаях перед `>>>` вставлена дополнительная пустая строка, для повышения удобочитаемости печатной версии книги. Эти пустые строки не нужно на самом деле вводить в интерактивной командной строке:

30 Об этой книге

```
>>> print("Hello, world!")  
Hello, world!
```

Этой пустой строки не будет во время
настоящего интерактивного сеанса

```
>>> print("Until next time...")  
Until next time...
```

Мы также активно применяем блокноты Jupyter, как описывается в главе 1, в подразделе 1.5.1. Включенный в официальный репозиторий GitHub код из блокнотов выглядит следующим образом:

```
# In[1]:  
print("Hello, world!")  
  
# Out[1]:  
Hello, world!  
  
# In[2]:  
print("Until next time...")  
  
# Out[2]:  
Until next time...
```

Практически все наши блокноты с примерами содержат следующий стереотипный код в первой ячейке (в первых главах некоторые его строки могут отсутствовать), которые мы в дальнейшем не будем включать в печатную версию книги:

```
# In[1]:  
%matplotlib inline  
from matplotlib import pyplot as plt  
import numpy as np  
  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
import torch.optim as optim  
  
torch.set_printoptions(edgeitems=2)  
torch.manual_seed(123)
```

В остальном блоки кода представляют собой частичные или полные фрагменты исходных файлов .py.

Листинг 15.1. main.py:5, def main

```
def main():  
    print("Hello, world!")  
  
if __name__ == '__main__':  
    main()
```

Во многих примерах книги используются отступы в два пробела. Но вследствие ограничений печати листинги кода ограничиваются строками в 80 символов, что неудобно для фрагментов кода с большими отступами. Отступы в два пробела позволяют устранить излишние переносы строк. Во всем доступном для скачивания коде для этой книги (опять же по адресам <https://www.manning.com/books/deep-learning-with-pytorch> и <https://github.com/deep-learning-with-pytorch/dlwpt-code>) везде используются отступы в четыре пробела. Переменные, название которых оканчивается на `_t`, представляют собой тензоры, хранимые в памяти CPU, на `_g` заканчиваются тензоры, хранимые в памяти GPU, на `_a` — массивы NumPy.

Аппаратные и программные требования

Для изучения части I не требуется особых вычислительных ресурсов. Достаточно любого более или менее современного компьютера или облачного сервиса. Аналогично не требуется какой-либо конкретной операционной системы. В части II мы предполагаем, что выполнение полного обучения для более продвинутых примеров потребует GPU с поддержкой CUDA. Используемые в части II параметры по умолчанию требуют GPU с 8 Гбайт памяти (рекомендуем использовать видеокарту NVIDIA GTX 1070 или вариант помощнее), но эти параметры можно откорректировать, если на вашем аппаратном обеспечении доступно меньше RAM. Исходные данные для проекта выявления рака из части II потребуют около 60 Гбайт трафика для скачивания, а для обучения в системе потребуется (как минимум) 200 Гбайт свободного места на диске. К счастью, работающие в интернете вычислительные сервисы недавно начали предлагать бесплатное время работы на GPU. Мы обсудим вычислительные требования подробнее в соответствующих разделах.

Вам понадобится Python 3.6 или более поздняя версия; инструкции можно найти на сайте Python (<http://www.python.org/downloads>). Информацию об установке PyTorch см. в руководстве Get Started на официальном сайте PyTorch (<https://pytorch.org/get-started/locally>). Мы рекомендуем пользователям Windows устанавливать его с помощью Anaconda или Miniconda (<https://www.anaconda.com/distribution> или <https://docs.conda.io/en/latest/miniconda.html>). В других операционных системах, например Linux, обычно имеется намного больше удобных вариантов, из которых Pip — самая распространенная система управления пакетами для Python. Мы предоставляем файл `requirements.txt` для установки зависимостей с помощью Pip. Поскольку современные ноутбуки Apple не включают GPU с поддержкой CUDA, заранее скомпилированные пакеты macOS для PyTorch работают только с CPU. Конечно, опытные пользователи могут устанавливать пакеты так, как удобнее для совместимости с их любимой средой разработки.

Прочие источники информации в интернете

Хотя для чтения этой книги не требуется никаких предварительных знаний глубокого обучения, она не является основательным введением в ГО. Мы

охватили в ней основы, но главной целью было научить читателя работать с библиотекой PyTorch. Мы призываем заинтересованных читателей выработать интуитивное понимание глубокого обучения до, во время либо после чтения этой книги. Замечательное средство для формирования прочной ментальной модели и интуитивного понимания механизмов, лежащих в основе глубоких нейронных сетей, — книга «Грокаем глубокое обучение» (<https://www.piter.com/book.phtml?978544611334>). Исчерпывающее введение и справочник вы найдете в книге *Deep Learning* Яна Гудфеллоу (Goodfellow) и др.¹ (<http://www.deeplearningbook.org/>).

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.

¹ Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение.

Об авторах

Большую часть своей профессиональной жизни Эли потратил на работу в различных стартапах в Кремниевой долине на различных должностях, начиная с разработчика программного обеспечения до технического директора. На момент публикации этой книги он работает с машинным обучением в индустрии беспилотных автомобилей.

Лука Антига работал исследователем в сфере биоинженерии в 2000-х годах, а последнее десятилетие играл роль одного из основателей и технического директора компании, занимающейся проектированием систем искусственного интеллекта. Он участвовал в нескольких проектах с открытым исходным кодом, включая ядро PyTorch.

Томас Виман — инструктор и консультант по машинному обучению и PyTorch, проживающий в Мюнхене, а также разработчик ядра PyTorch. Как обладатель докторской степени по математике, он не боится теории, но подходит с практической стороны, применяя ее к сложным вычислительным задачам.

Об иллюстрации на обложке

Обложку украшает рисунок *Kabardian* («Кабардинец») из книги Жака Грассе де Сан-Савье (Jacques Grasset de Saint-Sauveur) *Costumes Civils Actuels de Tous les Peuples Connus* («Наряды из разных стран»), опубликованной во Франции в 1788 году. Широкое разнообразие коллекции нарядов Грассе де Сан-Савье напоминает нам о том, насколько 200 лет назад регионы мира были уникальны и индивидуальны. В те времена по одежде человека можно было легко определить, откуда он, чем занимается и каков его социальный статус.

Стили одежды с тех пор изменились, и уникальность различных регионов угасла. Зачастую непросто отличить даже жителя одного континента от жителя другого, не говоря уже о городах, регионах и странах. Возможно, мы променяли культурное многообразие на более разнообразную личную жизнь и уж точно на более разнообразную и стремительную технологичную реальность.

В наше время, когда компьютерные книги так мало отличаются друг от друга, издательство Manning подчеркивает изобретательность и оригинальность компьютерного мира обложками книг, основанными на богатом разнообразии культурной жизни двухвековой давности, возвращенном иллюстрациями Жака Грассе де Сен-Савье.

Часть I

Основы PyTorch

Добро пожаловать в первую часть нашей книги. Именно в ней вам предстоит сделать свои первые шаги с фреймворком PyTorch и обрести базовые навыки, необходимые для понимания внутреннего устройства его самого и основанных на нем проектов.

В главе 1 мы познакомимся с PyTorch, разберемся, что он собой представляет и какие задачи решает, а также как он связан с другими фреймворками глубокого обучения. Глава 2 проведет для вас экскурсию по фреймворку PyTorch и позволит поэкспериментировать с моделями, предобученными для решения разных интересных задач. Глава 3 будет посерьезнее, так как в ней вы изучите основную структуру данных, используемую в программах PyTorch: тензор. В главе 4 вам предстоит еще одно путешествие, но на этот раз по способам представления данных из различных предметных областей в виде тензоров PyTorch. В главе 5 вы узнаете, как программа учится на примерах и как ей в этом помогает PyTorch. В главе 6 приводятся основные сведения о том, что такое нейронные сети и как создать нейронную сеть с помощью PyTorch. В главе 7 мы займемся решением простой задачи классификации изображений с помощью нейросетевой архитектуры. Наконец, глава 8 демонстрирует, как можно решить ту же задачу намного более остроумным образом: с помощью сверточной нейронной сети.

В конце части I у нас будет все, что нужно для решения с помощью PyTorch реальной задачи в части II.

Знакомство с глубоким обучением и библиотекой PyTorch

В этой главе

- ✓ Как глубокое обучение меняет подход к машинному обучению.
- ✓ Почему фреймворк PyTorch отлично подходит для глубокого обучения.
- ✓ Типичный проект глубокого обучения.
- ✓ Аппаратное обеспечение, которое понадобится вам, чтобы следить за ходом примеров.

Довольно расплывчатый термин *«искусственный интеллект»* (*artificial intelligence*) охватывает множество дисциплин, отличающихся колоссальным объемом исследований, критики, путаницы, невероятной шумихи и нагнетания паники. Реальное положение дел, разумеется, выглядит намного оптимистичнее. Было бы нечестно утверждать, что современные машины учатся «думать» в каком-либо человеческом смысле этого слова. Просто мы открыли общий класс алгоритмов, способных очень-очень эффективно аппроксимировать сложные, нелинейные процессы, и их можно использовать для автоматизации задач, которые ранее могли выполнять только люди.

Например, на сайте <https://talktotransformer.com/> языковая модель GPT-2 может генерировать связные абзацы текста, по слову за раз. Если подать на ее вход текущий абзац, она сгенерирует следующее:

Далее мы планируем подать на вход список фраз из корпуса адресов электронной почты и посмотреть, сможет ли программа произвести синтаксический разбор этих списков как предложений. Опять же это намного запутаннее и сложнее, чем поиск в начале этого сообщения, но надеемся, что оно поможет вам разобраться в основах формирования структур предложений на различных языках программирования.

Достаточно связный текст для машины, даже если никакой общей идеи в этом повествовании нет.

Еще больше впечатляет то, что умение выполнять эти ранее доступные только людям задачи достигается машиной посредством обучения на примерах данных, а не программируется человеком через набор рукотворных правил. В каком-то смысле становится ясно, что не следует путать интеллект с самосознанием и что самосознание явно не требуется для успешного выполнения подобных задач. В конце концов, вопрос машинного интеллекта может оказаться не столь важным. Эдсгер В. Дейкстра (Edsger W. Dijkstra) обнаружил, что вопрос того, способны ли машины думать, «так же разумен, как и вопрос о том, способны ли подлодки плавать»¹.

Этот общий класс алгоритмов относится к подкатегории *глубокого обучения* (*deep learning*) ИИ, связанной с обучением на наглядных примерах математических сущностей, называемых *глубокими нейронными сетями* (*deep neural networks*). Глубокое обучение использует большие объемы данных для аппроксимации сложных функций, входные и выходные данные которых сильно отличаются (например, изображение на входе и строка текста с описанием этого изображения на выходе; или рукопись на входе и читающий их естественно звучащий голос на выходе; или, еще проще, связывание изображения золотистого ретривера с флагом, сообщающим: «Да, здесь есть золотистый ретривер»). Подобные возможности позволяют создавать программы с функциональностью, доступной ранее исключительно людям.

1.1. РЕВОЛЮЦИЯ ГЛУБОКОГО ОБУЧЕНИЯ

Чтобы оценить по достоинству смену парадигмы, которую несет этот подход, взглянем на картину в целом. До прошлого десятилетия под *машинным обучением* понимался более широкий класс систем, активно использующих так называемое *проектирование признаков* (*feature engineering*). Признаки представляют собой преобразования входных данных, которые нужны последующим алгоритмам, например классификаторам, для выдачи правильных результатов на новых данных. Проектирование признаков состоит в том, чтобы

¹ Dijkstra E. W. The Threats to Computing Science. <http://mng.bz/nPJ5>.

найти правильные преобразования для успешного решения поставленной задачи последующим алгоритмом. Например, чтобы отличить единицы от нулей на изображениях рукописных цифр, можно придумать набор фильтров для оценки направлений границ на изображениях, а затем обучить классификатор предсказывать правильную цифру по распределению этих направлений. Еще один удобный признак — количество замкнутых контуров, как у цифр 0 и 8, и особенно в петлях цифры 2.

Глубокое обучение же, с другой стороны, нацелено на автоматический поиск подобных признаков из исходных необработанных данных для успешного решения задачи. В примере с нулями и единицами фильтры обновляются во время обучения с помощью итеративного анализа пар примеров данных и целевых меток. Это не означает, что проектирование признаков не используется при глубоком обучении: зачастую требуется внести какие-либо априорные знания в обучающуюся систему. Однако именно возможности ввода и обработки данных с последующим выделением полезных представлений на основе примеров данных составляют наиболее сильную сторону нейронных сетей. Специалисты по глубокому обучению стремятся в основном не создавать эти представления вручную, а работать с математической сущностью, которая бы извлекала представления из обучающих данных самостоятельно. Зачастую эти автоматически генерируемые признаки лучше созданных вручную! Как и в случае многих других революционных технологий, этот факт привел к изменению самой концепции.

На рис. 1.1 показан специалист, занятый описанием признаков и вводом их в алгоритм обучения, и результаты его работы будут зависеть от того, насколько хорошо подобраны функции. Справа показано глубокое обучение, при котором необработанные данные подаются на вход алгоритма, автоматически выделяющего признаки по степени их влияния на качество решения задачи, и результаты будут зависеть от способности специалиста направить алгоритм в сторону нужной цели.

Начиная с правой стороны рис. 1.1, мы уже получаем представление, что нужно для успешного выполнения глубокого обучения.

- Способ ввода и предварительной обработки имеющихся данных.
- Возможность каким-то образом описать глубоко обучающийся автомат.
- Обязательно нужен автоматизированный способ, *обучение* для получения полезных представлений и в конечном итоге желаемых результатов.

Очевидно, что нужно взглянуть повнимательнее на обучение, о котором мы столько говорим. Численная оценка расхождения между желаемыми и фактическими результатами модели (по принятому соглашению, чем меньше, тем лучше) основывается на *критерии*, вещественнозначной функции выходного сигнала модели и контрольных данных. Обучение представляет собой минимизацию

значений критерия с помощью постепенного улучшения глубоко обучающегося автомата до тех пор, пока не будут достигнуты низкие значения критерия даже на тех данных, которые не встречались во время обучения.

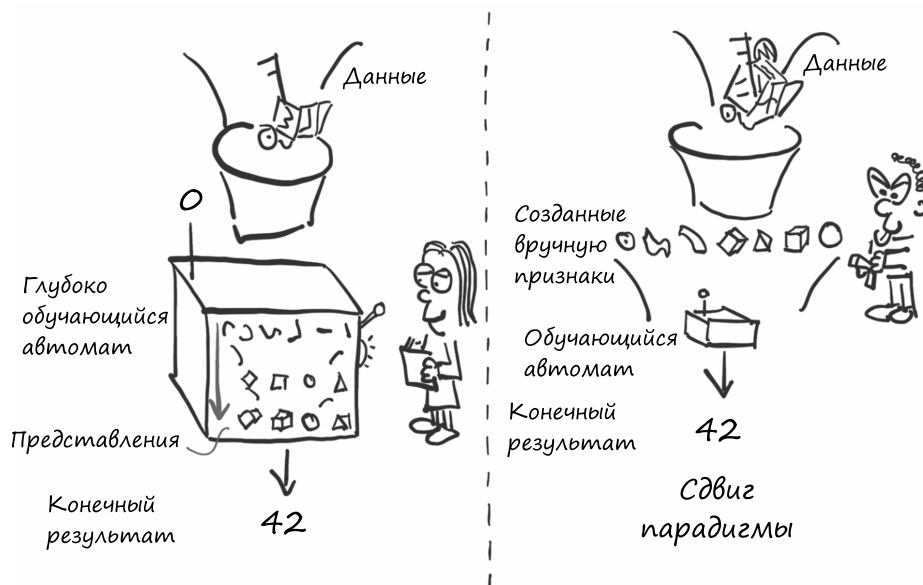


Рис. 1.1. Глубокое обучение исключает необходимость в создании признаков вручную, но требует больше данных и вычислительных ресурсов

1.2. ИСПОЛЬЗОВАНИЕ PYTORCH ДЛЯ ГЛУБОКОГО ОБУЧЕНИЯ

PyTorch — фреймворк/библиотека для программ на языке Python, который помогает создавать проекты глубокого обучения с упором на гибкость и возможность выражать модели глубокого обучения в характерном для Python стиле. Благодаря доступности и легкости использования PyTorch быстро нашел приверженцев в научном сообществе и за прошедшие с момента первого выпуска годы стал одним из самых значимых инструментов глубокого обучения с широким спектром приложений.

Как и сам Python для программирования вообще, PyTorch очень хорошо подходит для знакомства с глубоким обучением. В то же время библиотека PyTorch доказала свою пригодность для коммерческого использования для масштабных реальных задач. Мы убеждены, что понятный синтаксис, потоковый API и удобство отладки библиотеки PyTorch делают ее прекрасным вариантом для

знакомства с глубоким обучением. Мы настоятельно рекомендуем вам начать изучение библиотек глубокого обучения именно с PyTorch. Окажется ли она последней из изученных вами библиотек глубокого обучения — решать вам.

По существу, глубоко обучающийся автомат на рис. 1.1 представляет собой сложную математическую функцию, преобразующую входной сигнал в выходной. Для упрощения описания этой функции PyTorch предоставляет основополагающую структуру данных, *тензор* — многомерный массив, во многом схожий с массивами NumPy. Основываясь на этом фундаменте, PyTorch предоставляет возможности для выполнения быстрых математических операций на специализированном аппаратном обеспечении, что упрощает проектирование нейросетевых архитектур и обучение их на отдельных машинах или распараллеленных вычислительных ресурсах.

Эта книга задумывалась как отправной пункт для свободно владеющих Python разработчиков, специалистов по обработке данных и заинтересованных студентов, желающих научиться создавать с помощью PyTorch проекты глубокого обучения. Мы старались сделать ее как можно более доступной и полезной и надеемся, что вы сможете применить ее идеи в других предметных областях. Для этого мы выбрали практический подход и призываем вас держать свой компьютер наготове, чтобы экспериментировать с примерами и расширять их. Надеемся, что к тому моменту, когда вы закончите читать эту книгу, вы сможете взять источник данных и построить на его основе проект глубокого обучения с помощью превосходной официальной документации.

Хоть мы и делаем акцент на практических аспектах создания систем глубокого обучения с помощью PyTorch, мы надеемся, что подобное доступное введение в основы этого инструмента глубокого обучения станет для вас чем-то большим, чем просто способом обрести новые профессиональные навыки. Это шаг к тому, чтобы вооружить новое поколение ученых, инженеров и специалистов-практиков из разных сфер практическими знаниями, которые станут основой многих программных проектов в следующие десятилетия.

Чтобы вы могли извлечь из этой книги максимальную пользу, вам понадобятся:

- хотя бы небольшой опыт программирования на языке Python. Мы не станем на этом останавливаться: вы должны быть знакомы с типами данных Python, классами, числами с плавающей запятой и т. д.;
- желание учиться и погрузиться в работу с головой. Мы начнем с основ и будем постепенно набирать практические знания, и вам будет гораздо проще учить материал, следуя за ходом примеров.

Как уже говорилось, книга разбита на три отдельные части. Часть I охватывает основы и подробно исследует возможности, предоставляемые PyTorch для воплощения в жизнь схемы с рис. 1.1. Часть II посвящена комплексному проекту,

касающемуся медицинских снимков (поиск и классификация опухолей на КТ-снимках) и основанному на описанных в части I базовых понятиях с добавлением более продвинутых. Часть III завершает книгу кратким обзором возможностей PyTorch по развертыванию моделей глубокого обучения в промышленной эксплуатации.

Глубокое обучение — обширная сфера. В книге мы охватим лишь крошечную его долю, а именно применение PyTorch для более узких проектов классификации и сегментирования с обработкой двумерных и трехмерных изображений в качестве поясняющих примеров. Основное внимание в издании уделяется практической стороне использования PyTorch, а основная цель — охватить объем материала, достаточный для последующего решения вами реальных задач машинного обучения с помощью глубокого обучения, например, в области машинного зрения или изучения новых моделей по мере их появления в научной литературе. Большинство, если не все последние свежие публикации по исследованиям в сфере глубокого обучения можно найти в репозитории препринтов arXiv, расположенном по адресу <https://arxiv.org/>¹.

1.3. ПОЧЕМУ PYTORCH?

Как мы уже говорили, глубокое обучение позволяет решать очень широкий спектр сложных задач, таких как машинный перевод, стратегические игры или идентификация объектов на загроможденных изображениях, путем обучения модели на наглядных примерах данных. Для практической реализации необходимы гибкие (подходящие для широкого диапазона задач) и эффективные (для обучения на больших объемах данных за разумное время) инструменты. Кроме того, обученная модель должна правильно работать при изменчивых входных данных. Давайте обсудим, по каким причинам мы решили использовать PyTorch.

Библиотеку PyTorch можно смело рекомендовать благодаря ее простоте. Многие ученые и специалисты-практики отмечают легкость в изучении, использовании, расширении и отладке приложений PyTorch. Фреймворк полностью отражает стиль Python и, несмотря на имеющиеся различия в тонкостях и практическом исполнении, в целом покажется знакомым разработчикам, которые ранее использовали Python.

Если точнее, программирование глубоко обучающегося автомата на PyTorch происходит совершенно естественным образом. PyTorch предоставляет тип данных `Tensor` для хранения чисел, векторов, матриц и вообще массивов, а также функции для работы с ними. Писать программы с ними можно поэтапно, и при

¹ Мы также рекомендуем сайт www.arxiv-sanity.com для упорядочения интересных вас научных статей.

желании интерактивно, как и в случае Python. Что будет вполне привычно знакомым с NumPy.

Но PyTorch предоставляет также возможности, благодаря которым особенно хорошо подходит для глубокого обучения: во-первых, ускорение вычислений, порой 50-кратное по сравнению с выполнением того же вычисления на CPU, с помощью графических процессоров (GPU). Во-вторых, поддержку численной оптимизации общих математических выражений, используемых глубоким обучением для своих задач. Обратите внимание, что обе эти возможности полезны для научных вычислений в целом, а не только для глубокого обучения. На самом деле вполне можно назвать PyTorch высокопроизводительной библиотекой Python с поддержкой оптимизации научных вычислений.

Определяющий фактор PyTorch — это выразительность, позволяющая разработчику реализовать запутанные модели без привнесения библиотекой излишней сложности. PyTorch обеспечивает едва ли не самый гладкий перевод идей в код на языке Python в сфере глубокого обучения. Именно поэтому PyTorch так широко применяют в научной сфере, о чем свидетельствует большое число его упоминаний на международных конференциях¹.

PyTorch также прекрасно подходит для перехода от исследований и разработки к использованию для практических задач. Несмотря на то что изначально PyTorch был ориентирован на научные изыскания, она снабжена высокопроизводительной средой выполнения C++, подходящей для развертывания моделей для выполнения вывода без использования Python, и может применяться для проектирования и обучения моделей на C++. Также PyTorch обогатился привязками к другим языкам программирования и интерфейсом для развертывания на мобильных устройствах. Эти возможности позволяют использовать гибкость PyTorch, в то же время разворачивая приложения там, где недоступна полноценная среда выполнения Python либо накладные расходы на нее оказались бы недопустимо высоки.

Конечно, заявить о простоте использования и высокой производительности легче всего. Надеемся, в процессе чтения вы согласитесь, что наши утверждения вполне обоснованны.

1.3.1. Общая картина сферы глубокого обучения

Хотя все аналогии несовершенны, похоже, что выпуск PyTorch 0.1 в январе 2017 года отметил переход от взрывного роста количества библиотек, адаптеров и форматов обмена данными к эпохе объединения и унификации.

¹ На Международной конференции по усвоению представлений (International Conference on Learning Representations, ICLR) 2019.Ю PyTorch был упомянут в 252 статьях, по сравнению с 87 в предыдущем году, практически наравне с TensorFlow (266 упоминаний).

ПРИМЕЧАНИЕ

Сфера глубокого обучения меняется в последнее время так стремительно, что ко времени выхода печатного издания этой книги информация в ней, вероятно, уже устареет. Ничего страшного, если вы не знакомы с некоторыми из упомянутых ниже библиотек.

На момент выхода первой бета-версии PyTorch:

- Theano и TensorFlow были первыми низкоуровневыми библиотеками, в которых пользователь мог описать граф вычислений для модели и выполнить его;
- Lasagne и Keras были высокоуровневыми обертками для Theano, причем Keras служил оберткой для TensorFlow и CNTK;
- различные ниши в экосистеме заполняли Caffe, Chainer, DyNet, Torch (основанный на Lua предшественник PyTorch), MXNet, CNTK, DL4J и др.

За последующие примерно два года картина изменилась кардинально. Разработчики объединились вокруг либо PyTorch, либо TensorFlow, а другие библиотеки стали использовать реже, за исключением заполнявших определенные особые ниши. Если кратко, то:

- активная разработка Theano, одного из первых фреймворков глубокого обучения, фактически прекратилась;
- TensorFlow:
 - полностью поглотил Keras, превратив последний в полноценный API;
 - предоставил режим немедленного выполнения, в чем-то напоминающий подход PyTorch к вычислениям;
 - выпустил TF 2.0, в котором режим немедленного выполнения использовался по умолчанию;
- разработанная компанией Google независимо от TensorFlow библиотека JAX начала укреплять позиции как эквивалент NumPy, только с поддержкой GPU, автоматического вычисления градиентов и JIT;
- PyTorch:
 - поглотил Caffe2 в качестве прикладной части;
 - заменил большинство низкоуровневого кода, унаследованного из основанного на Lua проекта Torch;
 - добавил поддержку ONNX — платформонезависимого формата описания моделей и обмена ими;
 - добавил среду отложенного выполнения (graph mode) *TorchScript*;
 - выпустил версию 1.0;
 - заменил CNTK и Chainer на фреймворки от корпоративных спонсоров.

TensorFlow отличается надежным механизмом для практического использования, обширным сообществом разработчиков во всех отраслях индустрии и хорошо известен на рынке. PyTorch завоевал огромную популярность в сферах науки и образования благодаря удобству использования и с тех пор набирает силу, поскольку исследователи и выпускники обучают ему студентов, а также переходят в коммерческий сектор. Также он наращивает обороты в смысле решений, находящихся в промышленной эксплуатации. Интересно, что с появлением TorchScript и режима немедленного выполнения наборы функций PyTorch и TensorFlow начинают приближаться друг к другу, хотя представление этих функций и общее впечатление от работы с ними сильно различаются.

1.4. ОБЗОР СРЕДСТВ ПОДДЕРЖКИ БИБЛИОТЕКОЙ PYTORCH ПРОЕКТОВ ГЛУБОКОГО ОБУЧЕНИЯ

Мы уже упоминали мельком несколько основных «кирпичиков» PyTorch. Давайте потратим немного времени и составим укрупненную схему основных компонентов PyTorch. Удобнее всего для этого проанализировать, что необходимо проекту глубокого обучения от PyTorch.

Во-первых, PyTorch содержит в названии Py, как в Python, но в нем очень много написанного не на Python кода. Собственно, из соображений быстродействия большая часть PyTorch написана на C++ и CUDA (<https://www.geforce.com/hardware/technology/cuda>), C++-подобном языке программирования от NVIDIA, позволяющем компилировать программы для массово-параллельного выполнения на GPU. К PyTorch можно непосредственно обращаться из C++, и мы обсудим это в главе 15. Эта возможность, в частности, появилась, чтобы предоставить надежную стратегию развертывания моделей в производстве. Впрочем, большую часть времени мы будем работать с PyTorch из Python, создавая модели, обучая их и решая с помощью обученных моделей настоящие задачи.

И действительно, именно в API Python библиотека PyTorch проявляет себя наилучшим образом в смысле удобства использования и интеграции с остальной экосистемой Python. Взглянем на общую модель, демонстрирующую, что представляет собой PyTorch.

Как мы уже упоминали, PyTorch, по своей сути, библиотека, предоставляющая *многомерные массивы (тензоры* в терминологии PyTorch) и обширную библиотеку операций над ними, предоставляемую модулем `torch`. Тензоры и операции над ними можно использовать как на CPU, так и GPU. Перенос вычислений с CPU на GPU в PyTorch требует всего одного-двух дополнительных вызовов функций. Вторая важнейшая составляющая — PyTorch предоставляет возможность тензорам отслеживать производимые над ними операции и аналитически вычислять производные входных сигналов вычислений относительно нужных

входных сигналов, что применяется при численной оптимизации и предоставляется тензорами естественным образом благодаря обработке «изнутри» с помощью движка *autograd* библиотеки PyTorch.

Благодаря тензорам и стандартной библиотеке операций над ними с автоматическим дифференцированием PyTorch может применяться в физике, визуализации, имитационном и обычном моделировании, а также во многих других областях — PyTorch используется самыми разнообразными способами в целом спектре научных приложений. Но PyTorch прежде всего библиотека глубокого обучения, а потому предоставляет пользователям все базовые элементы, необходимые для создания и обучения нейронных сетей. На рис. 1.2 приведена стандартная архитектура: загрузка данных, обучение модели и развертывание в дальнейшем этой модели в эксплуатации.

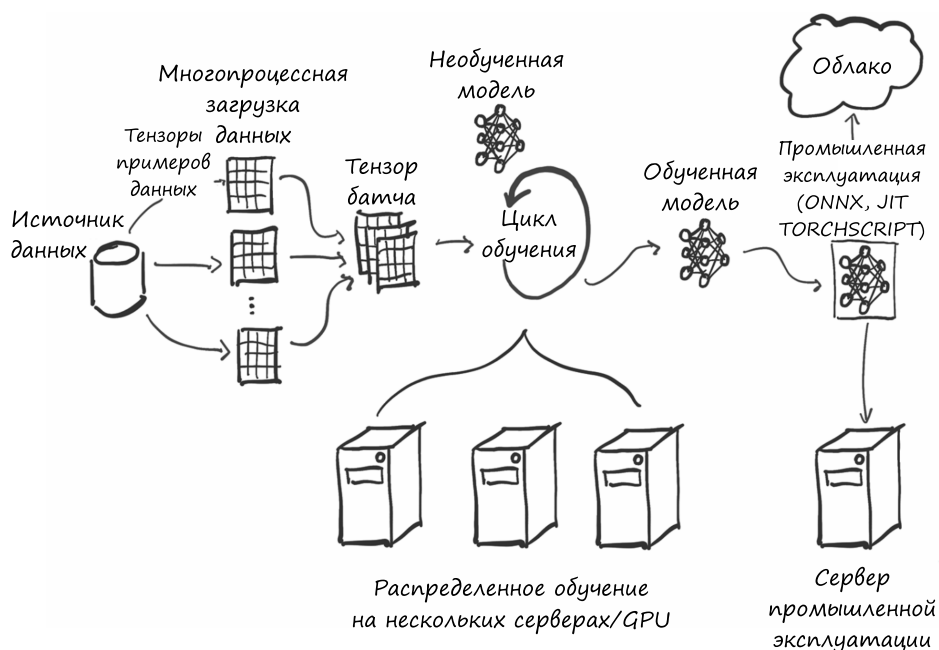


Рис. 1.2. Основная структура проекта PyTorch в укрупненном виде, с загрузкой данных, обучением и развертыванием в промышленной эксплуатации

Основные модули PyTorch, предназначенные для создания нейронных сетей, располагаются в `torch.nn`, включающем часто используемые слои нейронных сетей и прочие структурные компоненты. Здесь можно найти полносвязные слои, сверточные слои, функции активации и потерь (далее мы обсудим подробнее, что это все такое). Эти компоненты могут применяться для создания и задания начальных значений необученной модели, показанной в центре

рис. 1.2. Для обучения модели необходимо несколько дополнительных элементов: источник обучающих данных и оптимизатор, который бы подгонял модель к обучающим данным. Также необходимо каким-либо образом доставить модель и данные на аппаратное обеспечение, которое, собственно, и будет производить необходимые для ее обучения расчеты.

Из левой части рис. 1.2 видно, что еще до того, как обучающие данные попадут к модели, необходимо произвести немало вычислений¹. Прежде всего, необходимо физически получить данные, чаще всего из какого-либо хранилища, служащего их источником. Далее необходимо преобразовать все примеры из наших данных в нечто, с чем мог бы работать PyTorch: тензоры. Роль этого моста между пользовательскими данными (в каком бы они ни были формате) и тензорами PyTorch играет класс `Dataset`, включенный в модуль `torch.utils.data` библиотеки PyTorch. Поскольку данный процесс сильно отличается в разных задачах, нам придется реализовать получение данных из источника самостоятельно. В главе 4 мы рассмотрим подробнее, как представлять различные виды нужных нам данных в виде тензоров.

Хранилища данных обычно довольно медлительны, в частности, из-за задержки доступа, так что загрузку данных желательно распараллелить. Но, поскольку в число сильных сторон Python не входит удобная, эффективная, параллельная обработка, необходимо много процессов для загрузки данных, чтобы организовать их в *батчи*: тензоры, включающие несколько примеров данных. Это довольно замысловатая, но вместе с тем и относительно универсальная схема, и PyTorch предоставляет все необходимые возможности в классе `DataLoader`. Его экземпляры могут порождать дочерние процессы загрузки данных из объектов `Dataset` в фоновом режиме, так что данные будут готовы для использования в цикле обучения в любой момент. Мы познакомимся и поработаем с классами `Dataset` и `DataLoader` в главе 7.

Разобравшись с механизмом подготовки батчей примеров данных к использованию, мы можем обратить свое внимание на сам цикл обучения в центре рис. 1.2. Чаще всего цикл обучения реализуют в виде обычного цикла `for` Python. В простейшем случае модель выполняет необходимые вычисления на локальном CPU или отдельном GPU, и вычисления можно начинать сразу же с поступления данных в цикл обучения. Вполне возможно, что это будет ваша основная архитектура, и именно ее мы используем в этой книге.

На каждом шаге цикла обучения проверяется работа модели на полученных из загрузчика примерах данных, после чего выходные сигналы модели сравниваются с желаемыми (целевыми) на основе какого-либо *критерия* (*функции потерь*). PyTorch содержит не только компоненты для создания модели, но

¹ И это только выполняемая по ходу дела подготовка данных, а не предварительная обработка, которая может составлять немалую долю работы в реальных проектах.

и множество функций потерь на любой вкус. Они также доступны в модуле `torch.nn`. После сравнения фактических выходных сигналов с идеальными на основе функции потерь необходимо немного изменить модель, чтобы ее выходные сигналы больше напоминали целевые. Как уже упоминалось ранее, именно тут нам и пригодится механизм автоматического вычисления производных; но также понадобится и *оптимизатор*, доступный в модуле `torch.optim` PyTorch, который будет отвечать за обновление параметров. Мы займемся циклами обучения с функциями потерь и оптимизаторами в главе 5, а затем будем оттачивать наши навыки в главах с 6-й по 8-ю, прежде чем приступить к нашему большому проекту в части II.

Все чаще для этих целей используется различное специализированное аппаратное обеспечение, например несколько GPU или машин, объединяющих ресурсы для обучения большой модели, как показано в нижней части рис. 1.2. В подобных случаях можно воспользоваться подмодулями `torch.nn.parallel.DistributedDataParallel` и `torch.distributed` для работы с дополнительным аппаратным обеспечением.

Цикл обучения — это, возможно, самая скучная, но самая трудоемкая часть проекта глубокого обучения. По его завершении мы получаем модель, параметры которой оптимизированы под нашу задачу: обученную модель, показанную справа от цикла обучения на рис. 1.2. Модель, которая способна решить задачу, — это замечательно, но чтобы она принесла реальную пользу, ее нужно как-то разместить там, где она требуется. Эта часть — процесс *развертывания* — изображена в правой части рис. 1.2 и может включать в себя размещение модели на сервере или экспорт ее для загрузки в облако. Можно также интегрировать ее с большим приложением, либо запустить ее на смартфоне.

Одним из шагов развертывания может быть экспорт модели. Как уже упоминалось ранее, по умолчанию в PyTorch используется модель немедленного выполнения (*eager mode*). Как только интерпретатор Python выполняет инструкцию, связанную с PyTorch, базовая реализация C++ или CUDA сразу же производит соответствующую операцию. По мере того как все больше инструкций оперирует с тензорами, все больше операций выполняет базовая реализация.

PyTorch также предоставляет возможности предварительной компиляции моделей с помощью *TorchScript*. Используя TorchScript, PyTorch может преобразовать модель в набор инструкций, которые можно независимо вызывать из Python, допустим, из программ на C++ или на мобильных устройствах. Это можно считать своего рода виртуальной машиной с ограниченным набором инструкций, предназначенным для операций с тензорами. Экспортировать модель можно либо в виде TorchScript для использования со средой выполнения Python, либо в стандартизированном формате *ONNX*. Эти возможности формируют фундамент функциональности развертывания PyTorch, которую мы обсудим в главе 15.

1.5. АППАРАТНЫЕ И ПРОГРАММНЫЕ ТРЕБОВАНИЯ

Эта книга требует программирования и запуска задач, связанных с интенсивными численными вычислениями, например умножением большого количества матриц. Как оказалось, запустить предобученную сеть на новых данных вполне можно на любом современном ноутбуке или ПК. Даже повторное обучение небольшой части предобученной сети для ее адаптации к новому набору данных не требует специализированного аппаратного обеспечения. Вы можете повторить все, что мы делаем в части I этой книги, с помощью обычного персонального компьютера или ноутбука.

Однако мы предчувствуем, что завершение полного цикла обучения в более сложных примерах в части II потребует GPU с поддержкой CUDA. Используемые в части II параметры по умолчанию требуют GPU с 8 Гбайт RAM (рекомендуем использовать видеокарту NVIDIA GTX 1070 или мощнее), но эти параметры можно откорректировать, если на вашем аппаратном обеспечении доступно меньше RAM. Сразу оговоримся: без подобного аппаратного обеспечения можно обойтись, если вы готовы ждать, но выполнение на GPU сокращает время обучения как минимум на порядок (обычно в 40–50 раз). По отдельности необходимые для вычисления обновлений параметров операции занимают немного времени (от долей секунды до нескольких секунд) на современном аппаратном обеспечении наподобие CPU среднестатистического ноутбука. Проблема в том, что обучение требует выполнения этих операций снова и снова, много-много раз, с постепенным обновлением параметров сети для минимизации погрешности обучения.

Сети среднего размера могут потребовать от нескольких часов до нескольких дней для обучения с нуля на больших реальных наборах данных на рабочих станциях с хорошим GPU. Длительность обучения можно сократить за счет использования на одной машине нескольких GPU или даже еще сильнее — на кластере машин, оснащенных несколькими GPU. Подобные архитектуры не настолько недоступны, как может показаться, благодаря предложениям поставщиков облачных сервисов. DAWNBench (<https://dawn.cs.stanford.edu/benchmark/index.html>) — это интересная инициатива Стэнфордского университета, позволяющая оценивать время обучения и стоимость облачных сервисов для распространенных задач глубокого обучения на общедоступных наборах данных.

Так что если у вас будет GPU к моменту, когда вы доберетесь до части II, — замечательно. В противном случае рекомендуем вам изучить предложения различных облачных платформ, многие из которых предлагают блокноты Jupyter с поддержкой GPU и предустановленным PyTorch, зачастую в рамках бесплатного пакета услуг. Можно начать с Colaboratory (<https://colab.research.google.com>) от Google — это прекрасный вариант.

Осталось обсудить операционную систему (OS). PyTorch с первого выпуска поддерживал Linux и macOS, а в 2018-м начал поддерживать и Windows. Поскольку

современные ноутбуки Apple не оснащены GPU с поддержкой CUDA, заранее скомпилированные пакеты PyTorch под macOS работают только с CPU. В этой книге мы постараемся нигде не ориентироваться на использование конкретной операционной системы, хотя некоторые сценарии в части II показаны в варианте запуска из командной оболочки Bash в Linux. Команды этих сценариев можно легко преобразовать для использования в Windows. Для удобства код везде, где только возможно, будет показан так, как будто он выполняется из блокнота Jupyter.

Информацию об установке можно найти в руководстве Get Started на официальном сайте PyTorch (<https://pytorch.org/get-started/locally>). Мы рекомендуем пользователям Windows устанавливать PyTorch с помощью Anaconda или Miniconda (<https://www.anaconda.com/distribution> или <https://docs.conda.io/en/latest/miniconda.html>). В других операционных системах, например Linux, обычно имеется намного больше удобных вариантов, из которых Pip — самая распространенная система управления пакетами для Python. Мы предоставляем файл `requirements.txt` для установки зависимостей с помощью Pip. Конечно, опытные пользователи могут устанавливать пакеты так, как удобнее для совместимости с их любимой средой разработки.

Требования к полосе пропускания и пространству на диске в части II также довольно серьезны. Исходные данные для проекта обнаружения раковых опухолей из части II потребуют около 60 Гбайт трафика для скачивания, а в разархивированном виде займут на диске около 120 Гбайт. После разархивирования сжатые данные можно будет удалить. Кроме того, вследствие кэширования части данных из соображений быстродействия во время обучения потребуется еще 80 Гбайт. Всего для обучения в системе понадобится (как минимум) 200 Гбайт свободного места на диске. Хотя можно использовать и сетевое хранилище, но такое решение может отрицательно повлиять на скорость обучения, если доступ к сети происходит медленнее, чем к локальному диску. Желательно хранить данные на локальном SSD для быстрого их извлечения.

1.5.1. Блокноты Jupyter

Допустим, что вы установили PyTorch и прочие зависимости, а также проверили, что все работает. Ранее мы упоминали, как можно следить за ходом примеров в этой книге. Мы будем активно использовать для примеров кода блокноты Jupyter. Блокнот Jupyter выглядит как страница в браузере, из которой можно запускать код в диалоговом режиме. Этот код обрабатывается *ядром*, запущенным на сервере процессом, получающим код и возвращающим результаты, которые затем отображаются на странице. Блокнот хранит состояние ядра, например описанные в ходе выполнения кода переменные, в памяти до завершения или перезапуска его работы. Основная единица взаимодействия с блокнотом — *ячейка* (*cell*) — прямоугольное поле на странице, в которое можно вводить код для последующего выполнения

его ядром (запускаемого через пункт меню или с помощью сочетания клавиш Shift+Enter). В блокнот можно добавлять много ячеек, причем в новых ячейках будут доступны значения переменных, созданных в предыдущих. Возвращаемое последней строкой кода в ячейке значение выводится после выполнения сразу вслед за этой ячейкой, то же самое относится и к графикам. Комбинируя исходный код, результаты вычислений и текстовые ячейки в формате Markdown, можно создавать замечательные интерактивные документы. Узнать больше о блокнотах Jupyter вы можете на сайте этого проекта (<https://jupyter.org>).

Теперь вам нужно запустить сервер блокнотов из корневого каталога извлеченного из GitHub кода. Способ выполнения зависит от вашей операционной системы и того, как и куда вы установили Jupyter. Если у вас есть вопросы — смело задавайте их на форуме книги (<https://forums.manning.com/forums/deep-learning-with-pytorch>). После запуска сервера откроется окно используемого по умолчанию браузера со списком локальных файлов блокнотов.

ПРИМЕЧАНИЕ

Блокноты Jupyter — обладающий большими возможностями инструмент выражения различных идей в коде и их исследования. И если для нашей книги они подходят, то для других сценариев использования, возможно, нет. Мы полагаем, что важнее всего — устранить помехи и минимизировать лишние когнитивные затраты, и в разных случаях это происходит по-разному.

Весь проверенный код всех листингов из книги можно найти по адресу www.manning.com/books/deep-learning-with-pytorch, а также в репозитории GitHub: <https://github.com/deep-learning-with-pytorch/dlwpt-code>.

1.6. УПРАЖНЕНИЯ

1. Запустите интерактивную командную оболочку Python.
 - А. Какую версию Python вы используете? Надеемся, по крайней мере 3.6!
 - Б. Можете ли вы выполнить команду `import torch`? Какую версию PyTorch вы получили в результате?
 - В. Что вернула команда `torch.cuda.is_available()`? Соответствует ли ее результат ожидаемому, исходя из параметров вашего аппаратного обеспечения?
2. Запустите сервер блокнотов Jupyter.
 - А. Какую версию Python использует Jupyter?
 - Б. Совпадает ли расположение используемой Jupyter библиотеки `torch` с импортированной вами выше из интерактивной командной строки?

1.7. РЕЗЮМЕ

- Модели глубокого обучения автоматически обучаются связывать входные сигналы и желаемые выходные сигналы на примерах данных.
- Подобные PyTorch библиотеки дают возможность эффективно создавать и обучать нейросетевые модели.
- PyTorch минимизирует когнитивные затраты, уделяя особое внимание гибкости и быстродействию. По умолчанию в PyTorch используется немедленное выполнение операций.
- TorchScript позволяет заранее компилировать модели и вызывать их не только из Python, но и из программ на C++ и на мобильных устройствах.
- С момента выхода PyTorch в начале 2017 года экосистема инструментов глубокого обучения значительно усовершенствовалась.
- PyTorch предоставляет несколько вспомогательных библиотек для проектов глубокого обучения.

2

Предобученные сети

В этой главе

- ✓ Выполнение предобученных моделей распознавания изображений.
- ✓ Введение в GAN и CycleGAN.
- ✓ Модели описания изображений, способные генерировать их текстовые описания.
- ✓ Обмен моделями через Torch Hub.

Мы завершили первую главу обещанием открыть вам во второй потрясающие вещи, и теперь пришло время воплотить его в жизнь. Машинное зрение, безусловно, одна из сфер, на которую появление глубокого обучения оказало наибольшее влияние, и на это есть много причин. Была потребность в классификации или интерпретации содержимого естественных изображений, стали доступны очень большие наборы данных, были изобретены такие новые компоненты, как сверточные слои, работающие на GPU с невероятной точностью и скоростью. Все эти факторы объединились с желанием крупнейших интернет-компаний анализировать фотографии, снимаемые миллионами пользователей на мобильные устройства и размещаемые на платформах этих компаний. Идеальные условия.

Скоро мы научимся использовать результаты лучших исследователей в этой сфере, скачивая и запуская очень интересные модели, уже обученные на общедоступных

масштабных наборах данных. Предобученная нейронная сеть в чем-то подобна программе, которая принимает входные данные и формирует выходные. Поведение такой программы в плане желаемых пар «вход/выход» или нужных свойств определяется архитектурой нейронной сети и примерами, просмотренными ею во время обучения. Готовая модель — быстрый способ ускоренного запуска проекта глубокого обучения за счет использования опыта разработавших модель исследователей и вычислительного времени, затраченного на ее обучение.

В этой главе мы обсудим три часто встречающиеся предобученные модели: модель для маркировки изображения в соответствии с его содержимым, модель, способную формировать новое изображение на основе исходного, и модель, описывающую содержимое изображения простым английским языком. Вы научитесь загружать и запускать эти предобученные модели в PyTorch и познакомитесь с PyTorch Hub — набором инструментов, позволяющим с помощью единого интерфейса легко создавать модели PyTorch, подобные тем, которые мы будем обсуждать. А еще мы затронем источники данных, дадим определения таким терминам, как «метка» и, посетим родео с зебрами.

Если вы переходите на PyTorch с другого фреймворка глубокого обучения и хотели бы непосредственно заняться практикой, можете сразу читать главу 3. Здесь мы рассмотрим скорее развлекательные, чем серьезные вопросы, причем практически общие для всех утилит глубокого обучения. Это не значит, что они неважны! Но если вы уже работали с предобученными моделями в других фреймворках глубокого обучения, то прекрасно знаете, какие обширные возможности они открывают. А если вы уже знакомы с генеративными состязательными сетями (GAN), то вам не будет интересен наш подробный рассказ о них.

Впрочем, мы надеемся, что вы продолжите чтение этой главы, поскольку в ней описываются довольно важные навыки. Умение работать с предобученными моделями с помощью PyTorch — полезный навык и точка! Особенно он полезен, если модель обучена на большом наборе данных. Нам нужно будет привыкнуть к специфике получения и запуска нейронной сети на реальных данных, а также последующей визуализации и оценки результатов ее работы вне зависимости от того, обучали мы ее или нет.

2.1. ПРЕДОБУЧЕННЫЕ СЕТИ ДЛЯ РАСПОЗНАВАНИЯ ТЕМАТИКИ ИЗОБРАЖЕНИЯ

В качестве первого экскурса в глубокое обучение запустим современную глубокую нейронную сеть, предобученную на задаче распознавания образов. В репозиториях исходного кода доступно множество предобученных сетей. Исследователи часто публикуют свой исходный код вместе со статьями, причем нередко в код включаются весовые коэффициенты, полученные при обучении

модели на эталонном наборе данных. С помощью одной из таких моделей можно, например, без особых усилий оснастить веб-сервис возможностями распознавания изображений.

Предобученная сеть, которую мы здесь будем изучать, обучена на подмножестве набора данных ImageNet (<http://imagenet.stanford.edu/>). ImageNet — очень большой набор данных из более чем 14 миллионов изображений, поддерживаемый Стэнфордским университетом. Все его изображения маркированы при помощи иерархии существительных из набора данных WordNet (<http://wordnet.princeton.edu>) — большой лексической базы данных английского языка.

Набор данных ImageNet, подобно нескольким другим общедоступным наборам данных, появился в результате научных конкурсов. Такие конкурсы традиционно служат одними из главных площадок соревнования исследователей из различных учреждений и компаний. В частности, с момента своего появления в 2010 году большую популярность обрел конкурс крупномасштабных распознаваний зрительных образов ImageNet (ImageNet Large Scale Visual Recognition Challenge, ILSVRC). Этот конкретный конкурс включает в себя несколько заданий, меняющихся от года к году, например классификацию изображений (выяснение, какие категории объектов содержит изображение), локализацию объектов (определение местоположения объектов на изображении), обнаружение объектов (выявление и маркирование объектов на изображениях), классификацию обстановки (классификация общего плана на изображении) и разбор обстановки (разбиение изображения на области, относящиеся к различным семантическим категориям, например «корова», «дом», «сыр», «шляпа»). В частности, задача классификации изображений состоит в получении на основе входного изображения списка из пяти описывающих содержимое изображения меток (из общего списка в 1000 категорий), отсортированных по степени достоверности.

Обучающий набор данных для ILSVRC состоит из 1,2 миллиона изображений, маркированных одним из 1000 существительных (например, «собака») — *классов* изображений. Мы будем использовать далее в этом значении попеременно термины «метка» (*label*) и «класс» (*class*). На рис. 2.1 приведены некоторые из изображений ImageNet.

В конечном итоге мы хотим подавать свои собственные изображения на вход предобученной модели, как показано на рис. 2.2. В результате мы получим список прогнозируемых меток для этого изображения, по которым затем сможем понять, что наша модель думает о конкретном изображении. Предсказания для некоторых изображений безошибочны, а для других — нет!

Входное изображение сначала необходимо предварительно обработать, превратив в экземпляр класса многомерного массива `torch.Tensor`. Для изображения RGB с высотой и шириной тензор будет включать три измерения: три цветовых канала и два пространственных измерения заданного размера (в главе 3 мы обсудим подробнее, что представляет собой этот тензор, а пока что можете просто

считать его чем-то наподобие вектора или матрицы чисел с плавающей запятой). Наша модель принимает на входе это обработанное входное изображение и передает его предобученной сети, возвращающей оценки для всех классов. Наивысшая оценка соответствует наиболее вероятному классу в зависимости от веса. После этого каждый класс сопоставляется со своей меткой класса. Выходной сигнал содержится в `torch.Tensor` из 1000 элементов, каждый из которых соответствует оценке для конкретного класса.



Рис. 2.1. Небольшая выборка изображений ImageNet

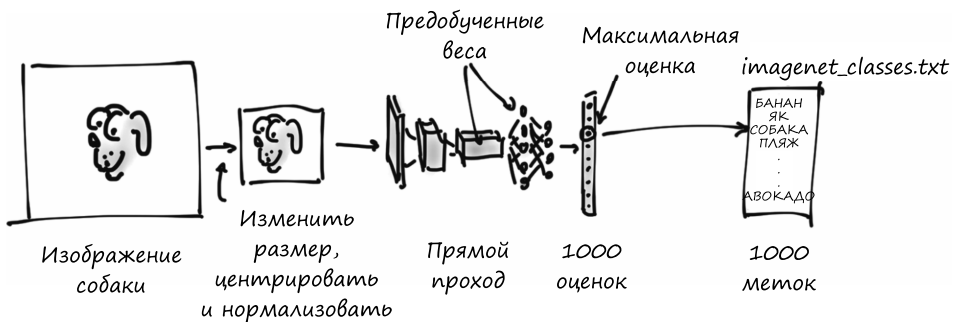


Рис. 2.2. Процесс выполнения вывода

Прежде чем заняться всем этим, необходимо получить саму нейронную сеть, заглянуть ей «под капот», чтобы понять, как она устроена, и разобраться с подготовкой данных для использования моделью.

2.1.1. Получение предобученной сети для распознавания изображений

Как мы уже упоминали, сейчас мы вооружимся нейронной сетью, обученной на ImageNet. Для этого взглянем на проект TorchVision (<https://github.com/pytorch/vision>), включающий несколько лучших нейросетевых архитектур, предназначенных для машинного зрения: AlexNet (<http://mng.bz/lo6z>), ResNet (<https://arxiv.org/pdf/1512.03385.pdf>) и Inception v3 (<https://arxiv.org/pdf/1512.00567.pdf>). Он также обеспечивает удобный доступ к таким наборам данных, как ImageNet, и прочим инструментам для работы с машинным зрением в PyTorch. Мы обсудим часть из них подробнее далее в этой книге. А пока что просто загрузим и запустим две сети: сначала AlexNet, одну из первых инновационных сетей для распознавания изображений; а затем остаточную сеть (residual network, сокращенно — ResNet), выигравшую, помимо прочего, в 2015 году конкурсы ImageNet по классификации, обнаружению и локализации. Если вы не установили и не настроили PyTorch в главе 1 — самое время это сделать.

Найти предобученные модели можно в `torchvision.models` (code/p1ch2/2_pre_trained_networks.ipynb):

```
# In[1]:
from torchvision import models
```

Взглянем на сами модели:

```
# In[2]:
dir(models)

# Out[2]:
['AlexNet',
 'DenseNet',
 'Inception3',
 'ResNet',
 'SqueezeNet',
 'VGG',
 ...
 'alexnet',
 'densenet',
 'densenet121',
 ...
 'resnet',
 'resnet101',
 'resnet152',
 ...
]
```

Названия, начинающиеся с заглавной буквы, относятся к классам Python, реализующим несколько популярных моделей, отличающихся архитектурой, то есть схемой операций между входом и выходом модели. Названия в нижнем регистре — вспомогательные функции, возвращающие созданные на основе этих классов модели, иногда с различными наборами параметров. Например, `resnet101` возвращает экземпляр ResNet со 101 слоем, `resnet18` содержит 18 слоев и т. д. Сначала мы займемся AlexNet.

2.1.2. AlexNet

В 2012 году архитектура AlexNet с большим отрывом от соперников выиграла конкурс ILSVRC с частотой ошибок топ-5 (коэффициент правильных ошибок) (то есть с наличием правильной метки в пяти лучших предсказаниях) в 15,4 %. Для сравнения: занявшая второе место архитектура, не основанная на нейронных сетях, показала результат лишь 26,2 %. Это был поворотный момент в истории машинного зрения: момент, когда сообщество стало осознавать потенциал глубокого обучения для задач машинного зрения. За этим прорывом последовал период непрерывного совершенствования, и теперь у наиболее современных архитектур и методов обучения частота ошибок топ-5 составляет всего 3 %.

По сегодняшним меркам AlexNet довольно небольшая сеть по сравнению с современными моделями. Но в нашем случае она прекрасно подходит, чтобы познакомиться с реально работающей нейронной сетью и научиться запускать ее предобученную версию для нового изображения.

Структура AlexNet приведена на рис. 2.3. Конечно, пока мы еще не знаем всего, что требуется для ее понимания, но кое-что можно сказать уже сейчас.

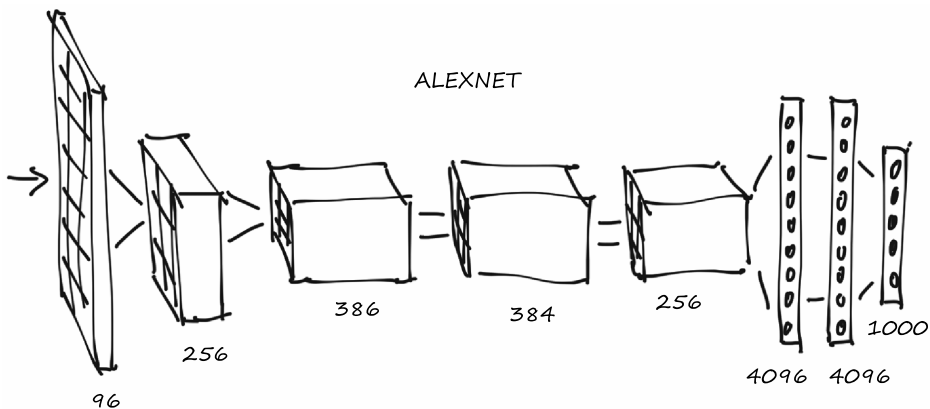


Рис. 2.3. Архитектура AlexNet

Прежде всего, каждый из блоков включает в себя набор операций умножения и сложения, а также небольшое количество прочих функций, как мы увидим в главе 5. Это можно считать своего рода фильтром — функцией, получающей на входе одно или несколько изображений и генерирующей в качестве выходного сигнала другие изображения. Конкретный способ определяется во время обучения на основе *просмотренных* моделью примеров данных, а также желаемых выходных сигналов для них.

На рис. 2.3 входные изображения поступают слева и проходят через пять комплектов фильтров, каждый из которых формирует некоторое количество выходных изображений. С каждым фильтром изображения уменьшаются в размере. Полученные последним комплектом фильтров изображения представляются в виде одномерного вектора из 4096 элементов и классифицируются, в результате чего генерируется 1000 выходных значений вероятности, по одному для каждого выходного класса.

Для запуска архитектуры AlexNet на каком-либо входном изображении необходимо создать экземпляр класса `AlexNet`. Вот таким образом:

```
# In[3]:
alexnet = models.AlexNet()
```

На этом этапе `alexnet` представляет собой объект, подходящий для выполнения архитектуры AlexNet. Понимание всех нюансов данной архитектуры нам сейчас не требуется. Пока что `alexnet` для нас представляет собой просто некий объект — «черный ящик», который можно запускать как функцию. Подав на вход `alexnet` входные данные четкого определенного размера, мы выполним прямой проход (*forward pass*) по сети, при котором входной сигнал пройдет через первый набор нейронов, выходные сигналы которых будут поданы на вход следующего набора нейронов, и так до самого итогового выходного сигнала. На практике это означает, что при наличии объекта `input` нужного типа можно произвести прямой проход с помощью оператора `output = alexnet(input)`.

Но если мы так поступим, то пропустим данные через всю сеть лишь для того, чтобы получить... мусор! А все потому, что сеть не была инициализирована: ее веса, числа, с которыми складываются и на которые умножаются входные сигналы, не были обучены на чем-либо, сеть сама по себе — чистый (или, точнее, *случайный*) лист. Необходимо либо обучить ее с нуля, либо загрузить веса, полученные в результате предыдущего обучения, что мы сейчас и сделаем.

Для этого вернемся к модулю `models`. Мы уже знаем, что названия в верхнем регистре соответствуют классам, реализующим популярные архитектуры, предназначенные для машинного зрения. С другой стороны, названия в нижнем регистре соответствуют функциям, создающим экземпляры моделей с заранее определенным количеством слоев и нейронов, а также, возможно, скачивающие и загружающие в них предобученные веса. Обратите внимание, что эти функции

несущественны, они просто упрощают создание экземпляра модели с соответствующим предобученной сети количеством слоев и нейронов.

2.1.3. ResNet

Сейчас с помощью функции `resnet101` мы создадим экземпляр сверточной нейронной сети из 101 слоя. Для наглядности: до появления остаточных нейронных сетей в 2015-м считалось, что добиться настолько устойчивого обучения при подобной глубине сети чрезвычайно сложно. Остаточные нейронные сети ухитрились сделать это возможным и тем самым побили несколько рекордов за один раз.

Создадим теперь экземпляр данной сети. Для этого мы передадим нашей функции аргумент, который укажет ей скачать веса `resnet101`, обученные на наборе данных ImageNet, включающем 1,2 миллиона изображений и 1000 категорий:

```
# In[4]:
resnet = models.resnet101(pretrained=True)
```

Пока мы смотрим на процесс загрузки, оцените тот факт, что `resnet101` может похвастаться 44,5 миллиона параметров — огромное количество для автоматической оптимизации!

2.1.4. На старт, внимание, почти что марш

Хорошо, что же мы только что сделали? Раз уж нам так интересно, можно взглянуть одним глазом, что представляет собой `resnet101`. Для этого можно вывести на экран значение возвращаемой модели и получить текстовое представление информации, аналогичной той, что мы видели в разделе 2.3, со всеми подробностями о структуре сети. На данный момент нам столько информации не нужно, но по мере чтения книги вы постепенно начнете понимать, что этот код нам говорит:

```
# In[5]:
resnet

# Out[5]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
    bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
    track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool1): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
    ceil_mode=False)
  (layer1): Sequential(
```

```

(0): Bottleneck(
...
)
)
(avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
(fc): Linear(in_features=2048, out_features=1000, bias=True)
)

```

Здесь мы видим `modules`, по одному на строку. Обратите внимание, что ничего общего с модулями Python у них нет: это отдельные операции, «кирпичики» нейронной сети. В других фреймворках глубокого обучения их также называют *слоями* (*layers*).

Если прокрутить вниз, можно увидеть множество модулей `Bottleneck`, один за другим (всего 101!), содержащих операции свертки и прочие модули. Именно так и устроена типичная глубокая нейронная сеть для машинного зрения: более или менее последовательный каскад фильтров и нелинейных функций, завершающийся слоем (`fc`), генерирующим оценки для каждого из 1000 выходных классов (`out_features`).

Переменную `resnet` можно вызывать как функцию, при этом она принимает на входе одно или несколько изображений и генерирует соответствующее количество оценок для каждого из классов ImageNet. Перед этим, впрочем, необходимо предварительно привести входные изображения к нужному размеру, а их значения (цвета) примерно в один числовой диапазон. Для этого модуль `torchvision` предоставляет преобразования (`transforms`), позволяющие быстро описывать конвейеры простейших операций предварительной обработки:

```

# In[6]:
from torchvision import transforms
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    ])

```

Здесь мы описали функцию `preprocess`, масштабирующую входное изображение до размера 256×256 , обрезающую его до 224×224 по центру, преобразующую в тензор (многомерный массив PyTorch: в данном случае трехмерный массив, содержащий цвет, высоту и ширину) и нормализующую его компоненты RGB (красный, зеленый, синий) до заданных среднего значения и стандартного отклонения. Если мы хотим получить от сети осмысленные ответы, все это должно соответствовать данным, полученным сетью во время обучения. Мы обсудим преобразования подробнее, когда будем создавать свои собственные модели распознавания изображений в подразделе 7.1.3.

Возьмем теперь изображение нашей любимой собаки (`bobby.jpg` из репозитория GitHub), проведем предварительную обработку и посмотрим, что о нем думает модель ResNet. Начнем с загрузки изображения из локальной файловой системы с помощью Pillow (<https://pillow.readthedocs.io/en/stable>) — модуля Python для обработки изображений:

```
# In[7]:  
from PIL import Image  
img = Image.open("../data/p1ch2/bobby.jpg")
```

Если вы работаете с блокнотом Jupyter, то, чтобы посмотреть в нем это изображение, необходимо выполнить следующую команду (изображение будет показано на месте `<PIL.JpegImagePlugin.JpegImageFile>`):

```
# In[8]:  
img  
# Out[8]:  
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=1280x720 at  
 0x1B1601360B8>
```

Либо можно вызвать метод `show`, при этом всплывет окошко с программой просмотра, в которой будет показано изображение с рис. 2.4.



Рис. 2.4. Бобби, наше особенное входное изображение

Далее можно пропустить это изображение через наш конвейер предварительной обработки:

```
# In[9]:  
img_t = preprocess(img)
```

При этом можно изменять размер изображения, обрезать его и нормализовывать входной тензор так, как нужно сети. Мы поговорим об этом подробнее в следующих двух главах; а пока что держитесь крепче:

```
# In[10]:
import torch
batch_t = torch.unsqueeze(img_t, 0)
```

Все готово к запуску модели.

2.1.5. Марш!

Процесс выполнения обученной модели на новых данных в сфере глубокого обучения называется *выводом (inference)*. Для выполнения вывода необходимо перевести сеть в режим `eval`:

```
# In[11]:
resnet.eval()

# Out[11]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
    bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
    track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
    ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
  ...
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

Если забыть сделать это, некоторые предобученные модели, например включающие *нормализацию по мини-батчам* и *дропаут*, не дадут никаких осмысленных результатов просто по причине их внутреннего устройства. Теперь, после установки режима `eval`, можно выполнять вывод:

```
# In[12]:
out = resnet(batch_t)
out
# Out[12]:
tensor([[ -3.4803, -1.6618, -2.4515, -3.2662, -3.2466, -1.3611,
          -2.0465, -2.5112, -1.3043, -2.8900, -1.6862, -1.3055,
  ...
          2.8674, -3.7442, 1.5085, -3.2500, -2.4894, -0.3354,
          0.1286, -1.1355, 3.3969, 4.4584]])
```

Только что было выполнено ошеломляющее количество операций с 44,5 миллиона параметров, был сгенерирован вектор с 1000 оценок, по одной для каждого класса ImageNet. Это не заняло много времени, не правда ли?

Теперь нужно определить метку класса, получившую максимальную оценку, и это расскажет нам о том, что модель увидела на рисунке. Если метка соответствует тому, как это изображение описал бы человек, — замечательно, значит, модель работает! Если же нет, значит, что-то пошло не так во время обучения, или изображение настолько отличается от ожидаемого моделью, что она не смогла обработать его должным образом, или возникла еще какая-либо аналогичная проблема.

Для просмотра списка предсказанных меток загрузим текстовый файл, в котором метки перечислены в том же порядке, в каком их видела сеть во время обучения, а затем возьмем метку, соответствующую максимальной оценке сети. Форма выходного сигнала практически всех моделей, предназначенных для распознавания изображений, аналогична той, с которой мы сейчас будем работать.

Загрузим файл с 1000 меток для классов набора данных ImageNet:

```
# In[13]:
with open('../data/p1ch2/imagenet_classes.txt') as f:
    labels = [line.strip() for line in f.readlines()]
```

Теперь нам нужно найти в полученном ранее тензоре `out` индекс, соответствующий максимальной оценке. Для этого воспользуемся функцией `max` PyTorch, возвращающей максимальное значение в тензоре, а также соответствующие ему индексы:

```
# In[14]:
_, index = torch.max(out, 1)
```

Сейчас можно получить доступ к метке по этому индексу. В данном случае `index` представляет собой не просто числовое значение Python, а состоящий из одного элемента одномерный тензор (а именно `tensor([207])`), так что необходимо получить числовое значение, которым мы могли бы воспользоваться в качестве индекса в нашем списке `labels` с помощью синтаксиса `index[0]`. Мы также воспользуемся `torch.nn.functional.softmax` (<http://mng.bz/BYnq>) для нормализации выходных сигналов к диапазону `[0, 1]` и деления на их сумму. В результате мы получим что-то вроде меры уверенности модели в конкретном предсказании. В данном случае модель на 96 % уверена, что видит перед собой золотистого ретривера:

```
# In[15]:
percentage = torch.nn.functional.softmax(out, dim=1)[0] * 100
labels[index[0]], percentage[index[0]].item()

# Out[15]:
('golden retriever', 96.29334259033203)
```

О-о, кто у нас тут хороший мальчик?

Поскольку модель генерирует оценки, мы можем также узнать занимающую второе место, третье и т. д. Для этого можно воспользоваться функцией `sort`, сортирующей значения в порядке возрастания или убывания, а также возвращающей индексы отсортированных значений в исходном массиве:

```
# In[16]:
_, indices = torch.sort(out, descending=True)
[(labels[idx], percentage[idx].item()) for idx in indices[0][:5]]

# Out[16]:
[('golden retriever', 96.29334259033203),
 ('Labrador retriever', 2.80812406539917),
 ('cocker spaniel, English cocker spaniel, cocker', 0.28267428278923035),
 ('redbone', 0.2086310237646103),
 ('tennis ball', 0.11621569097042084)]
```

Как видим, первые четыре — собаки (редбон — тоже порода собак, кто бы мог подумать?), после чего начинается странное. Пятый ответ, «теннисный мяч», возможно, возник потому, что существует столько фотографий теннисных мячей с собаками неподалеку, что модель фактически говорит нам: «С вероятностью 0,1 % я совершенно неправильно понимаю, что такое теннисный мяч». Это прекрасный пример принципиальных расхождений во взгляде на мир людей и нейронных сетей, а также насколько легко в наши данные могут закрасться странные, малозаметные систематические ошибки.

Пора поэкспериментировать! Подадим на вход нашей сети различные случайные изображения и посмотрим, что она нам вернет. Успешность работы сети во многом зависит от наличия соответствующих объектов в обучающем наборе данных. Если подать нейронной сети нечто выходящее за рамки обучающего набора данных, вполне возможно, что она достаточно уверенно вернет неправильный ответ.

Мы просто запустили сеть, выигравшую конкурс по классификации изображений в 2015 году. Она научилась на примерах различных собак узнавать нашу собаку, а также множество прочих объектов реального мира. Теперь мы узнаем, как различные архитектуры могут решать и прочие виды задач, начиная с генерации изображений.

2.2. ПРЕДОБУЧЕННАЯ МОДЕЛЬ, СОЗДАЮЩАЯ ВСЕ ЛУЧШИЕ ПОДДЕЛКИ

Представьте себе на минуту, что мы рецидивисты, планирующие заняться продажей поддельных «потерянных шедевров» великих художников.

Мы преступники, но не художники, так что специалисту сразу ясно, что наши фальшивые Рембрандты и Пикассо — любительские подделки, а не настоящие шедевры. Даже если мы хорошенько потренируемся и *сами* не сможем отличить созданное от настоящей картины, эксперты на местном аукционе сразу же нас вычислят. Хуже того, полученный ответ «Это явно подделка, убирайтесь отсюда» не поможет нам исправить дело! Нам придется случайным образом пробовать различные варианты, оценивая, в каких случаях выявление подделки занимает у экспертов *чуть* больше времени, и учесть эти особенности в наших будущих подделках. Подобный путь потребует слишком много времени.

Вместо этого необходимо найти историка искусства с достаточно гибкими моральными принципами, который бы изучил нарисованные нами картины и сказал в точности, что в них выдает подделку. Благодаря такой обратной связи мы сможем четко и ясно проложить путь к улучшению результатов модели, пока наш жуликоватый ученый не сможет отличить наших картин от настоящих.

Скоро наш «Боттичелли» будет висеть в Лувре, а их доллары — лежать в наших карманах. Мы разбогатеем!

И хотя это, конечно, немного шуточный сценарий, но лежащая в его основе технология вполне реальна и, вероятно, серьезно повлияет на субъективную достоверность цифровых данных в будущем. Вся концепция «фотографических свидетельств», возможно, безнадежно устареет, ведь так легко автоматизировать генерацию убедительных, хотя и поддельных, изображений и видео. Единственный ключевой ингредиент — данные. Давайте взглянем, как этот процесс происходит.

2.2.1. Игра GAN

Описанное выше в контексте глубокого обучения называется *игрой GAN* (the GAN game), при которой две сети, одна — художник, а вторая — историк искусства, стремятся перехитрить друг друга в создании и обнаружении подделок. GAN расшифровывается как *порождающая состязательная сеть* (generative adversarial network), где «*порождающая*» означает создание чего-либо (в данном случае поддельных шедевров), «*состязательная*» означает, что две сети соревнуются друг с другом, и, наконец, что такое «*сеть*» — вполне понятно. Этот вид сетей — один из самых интересных результатов недавних исследований в области глубокого обучения.

Напомним, что наша основная цель — генерация синтетических примеров класса изображений, в которых нельзя выявить подделку. Если смешать их с настоящими примерами данных, даже опытный эксперт не сможет различить, какие настоящие, а какие — подделки.

Роль художника в нашем сценарии играет сеть-генератор, задачей которой является генерация правдоподобно выглядящих картин, начиная с произвольной стартовой точки (входных данных). Сеть-дискриминатор — наш безразличный эксперт, задача которого — определить, сфабрикована ли картина генератором или относится к числу настоящих. Подобная схема с двумя сетями не типична для большинства архитектур глубокого обучения, но если реализовать на ее основе игру GAN, можно получить потрясающие результаты.

На рис. 2.5 показана примерная картина происходящего. Конечная цель генератора — «обмануть» дискриминатор и выдать ему поддельные картины за настоящие. Конечная цель дискриминатора — найти подделки, но также сообщить генератору, какие ошибки тот совершил при их генерации. В самом начале генератор генерирует маловразумительных трехглазых монстров, совершенно непохожих на рембрандтовские портреты. Дискриминатор легко отличает эту мешанину от настоящих картин. По мере обучения дискриминатор передает информацию генератору, благодаря которой последний постепенно совершенствует свои картины. К концу обучения генератор создает вполне убедительные подделки, а дискриминатор более не способен отличить их от настоящих картин.



Рис. 2.5. Общая картина игры GAN

Обратите внимание, что не следует понимать фразы «дискриминатор победил» и «генератор победил» буквально: никакого явного соревнования между ними не идет. Каждая из этих сетей просто обучается на результатах другой, оптимизируя тем самым параметры друг друга.

Эта методика позволила создавать генераторы, формирующие реалистично выглядящие изображения на основе одного шума и нормализованных сигналов, скажем атрибутов (например, в случае лиц: молодое, женское, с очками) другого изображения. Другими словами, хорошо обученный генератор усваивает адекватную модель генерации изображений, выглядящих правдоподобно даже для людей.

2.2.2. CycleGAN

Интересное развитие этой идеи — CycleGAN. CycleGAN способна превращать изображения из одной предметной области в изображения из другой (и обратно) без явно заданных пар соответствий в обучающем наборе данных.

На рис. 2.6 приведен технологический процесс CycleGAN для задачи превращения фотографии лошади в фотографию зебры и наоборот. Обратите внимание на наличие двух отдельных сетей-генераторов, а также двух отдельных дискриминаторов.

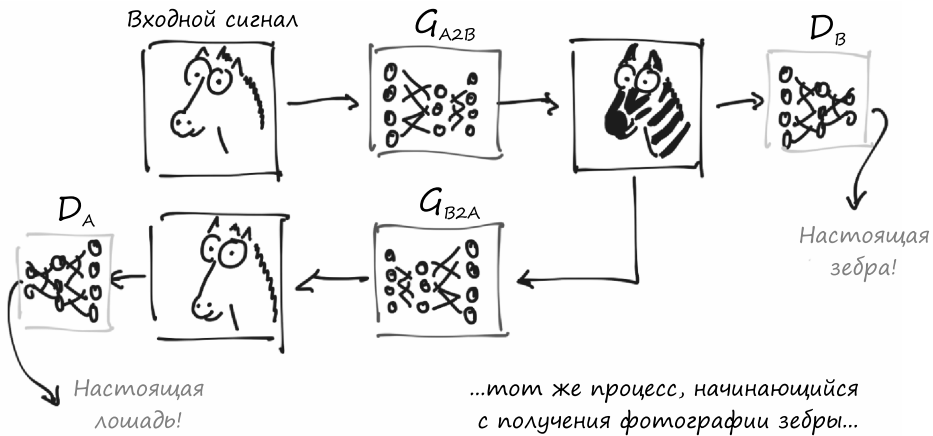


Рис. 2.6. CycleGAN обучается до такой степени, что может «обмануть» обе сети-дискриминатора

Как видно из рисунка, первый генератор учится создавать изображение, соответствующее целевому распределению (в данном случае зебрам), на основе изображения, относящегося к другому распределению (лошадям), так что дискриминатор не сможет определить, полученное из фотографии лошади изображение зебры — настоящая фотография зебры или нет. В то же время — и тут-то и играет свою роль префикс Cycle в названии — полученная поддельная фотография зебры проходит в обратном направлении (превращении зебры в лошадь в данном случае)

через другой генератор и проверяется на подлинность другим дискриминатором с другой стороны. Подобный цикл существенно повышает устойчивость процесса обучения, решая тем самым одну из изначальных проблем GAN.

Любопытно, что на этом этапе не требуются пары соответствий «лошадь/зебра» в качестве эталонных данных (попробуйте заставить их стоять так, чтобы позы совпадали). Генератору достаточно начать с набора несвязанных изображений лошадей и фотографий зебр, чтобы научиться делать свое дело, выходя таким образом за рамки чистой схемы обучения с учителем. Потенциал данной модели еще более широк: генератор обучается выборочно менять изображение объектов на кадре, без учителя. Никакой сигнал не указывает генератору, что гривы — это гривы, а ноги — это ноги, но все они преобразуются в соответствии с анатомией другого животного.

2.2.3. Сеть, превращающая лошадей в зебр

Можем поэкспериментировать с этой моделью прямо сейчас. Сеть CycleGAN уже обучена на наборе данных (несвязанных) изображений лошадей и зебр, извлеченных из набора данных ImageNet. Сеть усваивает, как преобразовать изображение одной или нескольких лошадей в зебр, внося в остальную часть изображения как можно меньше изменений. И хотя человечество не ждет тысячелетиями затаив дыхание инструмента, который позволил бы превращать лошадей в зебр, эта задача демонстрирует возможность обсуждаемых архитектур моделировать сложные процессы реального мира без непосредственного надзора. Несмотря на то что и у них есть свои ограничения, похоже, что в ближайшем будущем мы уже не сможем отличать настоящие фрагменты видеотрансляции от поддельных, что открывает настоящий ящик Пандоры, который мы пока что закроем от греха подальше.

Эксперименты с предобученной CycleGAN позволят нам изучить внимательнее, как устроена сеть — в данном случае генератор. Воспользуемся нашей старой доброй знакомой — ResNet. Здесь мы опустим описание класса `ResNetGenerator`. Соответствующий код приведен в первой ячейке файла `3_cyclegan.ipynb`, но реализация прямо сейчас нам неважна и может показаться вам слишком сложной, пока вы не наберетесь опыта работы с PyTorch. Пока нас интересует, на *что* она способна, а не *как* она это делает. Создадим экземпляр этого класса с параметрами по умолчанию (`code/p1ch2/3_cyclegan.ipynb`):

```
# In[2]:
netG = ResNetGenerator()
```

Мы создали модель `netG`, но весовые коэффициенты в ней пока что случайны. Мы уже упоминали ранее, что воспользуемся моделью генератора, предобученной на наборе данных `horse2zebra`, обучающие данные которой включают два набора из 1068 и 1335 изображений лошадей и зебр соответственно. Найти этот набор

Открываем файл с изображением лошади (рис. 2.7):

```
# In[7]:
img = Image.open("../data/p1ch2/horse.jpg")
img
```



Рис. 2.7. Всадник
на лошади.
Лошади не нравится

Видим какого-то парня на лошади (ненадолго, судя по картинке). В любом случае выполним предварительную обработку и превратим эту картинку в переменную нужной формы:

```
# In[8]:
img_t = preprocess(img)
batch_t = torch.unsqueeze(img_t, 0)
```

О нюансах пока не волнуйтесь. Главное, что мы видим общую картину. Теперь можно передать модели `batch_t`:

```
# In[9]:
batch_out = netG(batch_t)
```

`batch_out` — выходной сигнал генератора, который можно преобразовать обратно в изображение:

```
# In[10]:
out_t = (batch_out.data.squeeze() + 1.0) / 2.0
out_img = transforms.ToPILImage()(out_t)
# out_img.save('../data/p1ch2/zebra.jpg')
out_img

# Out[10]:
<PIL.Image.Image image mode=RGB size=316x256 at 0x23B24634F98>
```

О нет, ну кто же так ездит на зебре? Полученное изображение (рис. 2.8) не-идеально, но учтите, что сети непривычно видеть кого-то (условно) верхом на лошади. Стоит еще раз отметить, что процесс обучения не является обучением с учителем, при котором люди вручную распределяют десятки тысяч лошадей и вручную созданных в Photoshop полосок зебр. Генератор научился создавать изображения, способные обмануть дискриминатор, убедив последний, что речь идет об изображении зебры, причем изображение не внушает никаких подозрений (дискриминатор, конечно, никогда не бывал на родео).



Рис. 2.8. Всадник на зебре.
Зебре не нравится

С помощью состязательных сетей и прочих подходов было разработано немало других интересных генераторов. Одни способны создавать правдоподобные лица несуществующих людей, другие — превращать наброски в реалистично выглядящие картины воображаемых ландшафтов. Генеративные модели также применялись для генерации реалистично звучащих аудио, правдоподобного текста и приятной музыки. Вполне возможно, что на основе этих моделей в будущем будут создаваться вспомогательные инструменты для процесса творчества.

Если серьезно, то сложно переоценить потенциал подобных исследований. Возможности и распространенность утилит вроде той, которую мы только что скачали, будут только расти. В частности, внимание прессы сейчас приковано к технологии замены лиц на фотографиях. Поиск по ключевым словам *deep fake* выдаст множество примеров контента¹ (стоит отметить, что при этом находится

¹ Один из примеров описан в статье Vox Jordan Peele's simulated Obama PSA is a double-edged warning against fake news, Aja Romano; <http://mng.bz/dxBz> (предупреждаем: в статье присутствует обсценная лексика).

немало неподходящего для просмотра на работе контента; как и в прочих случаях в интернете, переходите по ссылкам осторожно).

До сих пор мы работали с моделью, исследующей изображения, и моделью, создающей новые изображения. В заключение мы рассмотрим модель, охватывающую еще один важнейший ингредиент: естественный язык.

2.3. ПРЕДОБУЧЕННАЯ СЕТЬ ДЛЯ ОПИСАНИЯ ОБСТАНОВКИ

Для знакомства с моделями, работающими с естественным языком, мы воспользуемся предобученной моделью описания изображений, любезно предоставленной Ротанем Ло (Ruotian Luo)¹. Она представляет собой реализацию модели NeuralTalk2 Андрея Карпати (Andrej Karpathy). Она генерирует для полученного на входе изображения описание обстановки на нем на английском языке, как показано на рис. 2.9. Данная модель обучена на большом наборе изображений с их описаниями в виде фраз: например, *A Tabby cat is leaning on a wooden table, with one paw on a laser mouse and the other on a black laptop*².



Рис. 2.9. Общая идея модели для описания изображений

Эта модель описания изображений состоит из двух связанных половинок. Первая из них — сеть, обучающаяся генерировать «информативные» численные

¹ Актуальный клон соответствующего кода можно найти по адресу <https://github.com/deep-learning-with-pytorch/ImageCaptioning.pytorch>.

² «Кошка лежит на деревянном столе, держа одну лапу на лазерной мышке, а другую — на черном ноутбуке». — *Примеч. пер.*

представления обстановки (кошка окраса табби, лазерная мышка, лапа), передаваемые в качестве входного сигнала второй половинке. Эта вторая половинка представляет собой *рекуррентную нейронную сеть* (*recurrent neural network*), генерирующую связное предложение путем объединения этих числовых описаний. Обе половинки модели обучаются вместе на парах «изображение/описание».

Вторая половинка модели называется *рекуррентной*, поскольку генерирует выходные сигналы (отдельные слова) при последовательных прямых проходах, входной сигнал каждого из которых включает выходные сигналы предыдущего прохода. При этом возникает зависимость следующего слова от сгенерированных ранее, как и можно ожидать при работе с предложениями и в целом с последовательностями.

2.3.1. NeuralTalk2

Модель NeuralTalk2 можно найти на <https://github.com/deep-learning-with-pytorch/ImageCaptioning.pytorch>. Можно поместить несколько изображений в каталог `data` и выполнить следующий сценарий:

```
python eval.py --model ./data/FC/fc-model.pth
➡ --infos_path ./data/FC/fc-infos.pkl --image_folder ./data
```

Попробуем сделать это для нашего изображения `horse.jpg`. Получаем: *A person riding a horse on a beach* («Человек едет на лошади по пляжу»). Вполне адекватное описание.

А теперь просто ради развлечения посмотрим, удастся ли нашей CycleGAN обмануть модель NeuralTalk2. Добавим в каталог `data` изображение `zebra.jpg` и запустим модель снова: *A group of zebras are standing in a field* («Группа зебр стоит в поле»). Животное угадано верно, но модель увидела на изображении несколько зебр, а не одну. Ясно, что модель вряд ли видела зебру или всадника на зебре (еще и с каким-то нетипичным рисунком на шкуре). Кроме того, вполне вероятно, что в обучающем наборе данных зебры изображались группами, отсюда и возможная систематическая ошибка. Описательная сеть также не упомянула всадника. И вероятно, по той же причине: в обучающем наборе данных не встречались всадники на зебрах. В любом случае результат впечатляет: мы сгенерировали поддельное изображение с невозможной ситуацией, и описательная сеть проявила достаточную гибкость, чтобы правильно понять, что изображено.

Хотелось бы подчеркнуть, что подобные достижения, практически невозможные до появления глубокого обучения, требуют всего лишь менее тысячи строк кода, универсальной архитектуры, не связанной с лошадьми или зебрами, и корпуса изображений с описаниями (в данном случае набор данных MS COCO). Никаких защитных критериев или грамматик — все, включая предложение с описанием, формируется на основе закономерностей в данных.

Архитектура сети в этом последнем случае несколько сложнее, чем предыдущие, так как включает две сети. Одна из них — рекуррентная, но создана из тех же «кирпичиков», предоставляемых PyTorch.

На момент написания данной книги подобные модели используются скорее в исследованиях и в качестве интересных новинок, а не испытанных стандартных методов. Их результаты, хотя и весьма многообещающие, для практического использования недостаточно хороши... пока что. Со временем (и с появлением дополнительных обучающих данных) следует ожидать, что с помощью моделей этого класса можно будет описывать слабовидящим людям окружающий мир, расшифровывать сцены из видео и решать другие аналогичные задачи.

2.4. TORCH HUB

Предобученные модели публиковались с первых же дней глубокого обучения, но до появления PyTorch 1.0 не существовало унифицированного интерфейса для пользователей. TorchVision — неплохой пример аккуратного интерфейса, как было показано ранее в этой главе; но другие авторы, как видно из CycleGAN и NeuralTalk2, предпочитают разные варианты архитектур.

В PyTorch 1.0 был добавлен Torch Hub — механизм публикации моделей в GitHub с предобученными весовыми коэффициентами и без них и с возможностью использования их через понятный PyTorch интерфейс. Благодаря ему загрузка сторонней предобученной модели стала такой же легкой, как загрузка модели TorchVision.

Автору, чтобы опубликовать модель через механизм Torch Hub, необходимо всего лишь поместить файл `hubconf.py` в корневой каталог репозитория GitHub. Структура этого файла очень проста:

```
dependencies = ['torch', 'math']

def some_entry_fn(*args, **kwargs):
    model = build_some_model(*args, **kwargs)
    return model

def another_entry_fn(*args, **kwargs):
    model = build_another_model(*args, **kwargs)
    return model
```

← Необязательный список модулей, от которых зависит данный код

← Одна или несколько функций, открываемых пользователям в качестве входных точек репозитория. Эти функции должны инициализировать модели в соответствии с аргументами и возвращать их

Теперь интересные предобученные модели можно искать в репозиториях GitHub, содержащих файл `hubconf.py`, зная сразу же, что их можно будет загрузить с помощью модуля `torch.hub`. Давайте взглянем, как сделать это на практике. Для этого мы вернемся к TorchVision, как ясному примеру взаимодействия с Torch Hub.

Заходим на страницу <https://github.com/pytorch/vision> и видим там файл `hubconf.py`. Прекрасно, с этим все в порядке. Теперь прежде всего необходимо изучить этот файл и найти точки входа для репозитория — нам понадобится указать их позднее. В случае TorchVision их две: `resnet18` и `resnet50`. Мы уже знаем, что они делают: возвращают 18-и 50-слойную модели ResNet соответственно. Также видим, что функции точек входа включают ключевой аргумент `pretrained`. При его значении `True` возвращаемые модели инициализируются усвоенными из ImageNet весовыми коэффициентами, как мы видели ранее в этой главе.

Итак, нам известны репозиторий, точки входа и один интересный ключевой аргумент. Вот и все, что требуется для загрузки модели с помощью модуля `torch.hub`, даже не нужно клонировать репозиторий. Да, именно так, PyTorch сделает это за нас:

```
import torch
from torch import hub

resnet18_model = hub.load('pytorch/vision:master',
                          'resnet18',
                          pretrained=True)
```

Приведенный код скачивает копию состояния ветки `master` репозитория `pytorch/vision`, вместе с весовыми коэффициентами в локальный каталог (по умолчанию `.torch/hub` в домашнем каталоге) и выполняет функцию точки входа `resnet18`, возвращающую созданный экземпляр модели. В зависимости от среды Python может пожаловаться на отсутствие какого-либо модуля, например `PIL`. Torch Hub не устанавливает отсутствующие зависимости, но сообщает о них нам, чтобы мы могли предпринять нужные действия.

На этом этапе можно вызвать возвращенную модель с соответствующими аргументами и выполнить прямой проход, как мы делали ранее. Теперь каждая модель, опубликованная через этот механизм, будет доступна нам с помощью тех же методов, как бы далеко она от нас ни находилась, и это очень приятная новость.

Обратите внимание: предполагается, что точки входа будут возвращать модели, но, строго говоря, они не обязаны это делать. Например, одна точка входа может производить преобразование входных данных, а другая — отвечать за преобразование выходных вероятностей в текстовые метки. Или можно создать точку входа для самой модели, а другую — для модели вместе с этапами предварительной обработки и постобработки. С помощью этого разработчики PyTorch обеспечили достаточную степень стандартизации и в то же время значительную гибкость. Далее мы увидим, к чему приводят эти возможности.

Torch Hub — достаточно новый инструмент на момент написания данной книги, и существует всего несколько моделей, опубликованных таким образом. Найти их можно, выполнив в поисковике запрос `github.com hubconf.py`. Надеемся, что

этот список в будущем вырастет, так как все новые авторы будут публиковать модели через этот канал.

2.5. ИТОГИ ГЛАВЫ

Надеемся, эта глава была интересной. Мы немного поэкспериментировали с моделями, созданными при помощи PyTorch и оптимизированными для выполнения конкретных задач. На самом деле самые предприимчивые читатели могли уже разместить некоторые из этих моделей на веб-сервере для коммерческого использования, разделив доходы с авторами моделей!¹ Разобравшись, как устроены эти модели, мы сможем воспользоваться полученными знаниями для скачивания предобученных моделей и быстрого перепрофилирования их под слегка отличающиеся задачи.

Нам также предстоит увидеть, как можно создавать модели для решения различных задач на различных видах данных с помощью одних и тех же «кирпичиков». Особенно удобно то, что PyTorch предоставляет эти «кирпичики» в форме базового набора инструментов: PyTorch не очень большая библиотека с точки зрения API, особенно по сравнению с другими фреймворками глубокого обучения.

В этой книге мы не стремимся изучить весь API PyTorch или все возможные архитектуры глубокого обучения, а просто хотим познакомиться с этими «кирпичиками» на практике. Благодаря такому фундаменту вы сможете работать с необходимой документацией и смотреть репозитории, доступные в интернете.

Начиная со следующей главы, мы отправимся в путешествие, в ходе которого сможем с помощью PyTorch научить с нуля наш компьютер навыкам, подобным описанным в этой главе. А также узнаем об эффективном способе решения задач в отсутствие большого числа точек данных — адаптации готовой предобученной сети к новым данным. Это еще одна причина, по которой предобученные сети так важны для специалистов по глубокому обучению. А далее поговорим о первом из основных «кирпичиков»: о тензорах.

2.6. УПРАЖНЕНИЯ

1. Подайте на вход модели преобразования лошадей в зебр наше изображение золотистого ретривера.
 - А. Какая предварительная обработка необходима для этого изображения?
 - Б. Как выглядит результат работы модели?

¹ Свяжитесь с авторами моделей и узнайте больше о франчайзинговых возможностях!

2. Поищите в GitHub проекты с файлом `hubconf.py`.
 - А. Сколько репозиторийев нашлось?
 - Б. Найдите какой-нибудь интересный проект с `hubconf.py`. Понятно ли назначение этого проекта из документации?
 - В. Запомните этот проект и вернитесь к нему, когда дочитаете эту книгу. Понятна ли вам его реализация?

2.7. РЕЗЮМЕ

- Предобученная сеть — модель, уже обученная на каком-либо наборе данных. Подобные сети обычно возвращают полезные результаты срезу после загрузки параметров сети.
- Благодаря умению пользоваться предобученными моделями не нужно проектировать или обучать нейронную сеть для ее интеграции в проект.
- AlexNet и ResNet — две глубокие сверточные сети, ставшие эталонными в сфере распознавания изображений сразу же после своего выхода в свет.
- Генеративные состязательные сети (GAN) состоят из двух частей: генератора и дискриминатора, совместно обеспечивающих генерацию неотличимых от оригиналов результатов.
- Архитектура CycleGAN поддерживает возможность преобразования (в обе стороны) между двумя различными классами изображений.
- Гибридная архитектура модели NeuralTalk2 позволяет генерировать текстовое описание полученного на входе изображения.
- Torch Hub — стандартизированный способ загрузки моделей и весовых коэффициентов из любого проекта, включающего соответствующий файл `hubconf.py`.

3

В начале был тензор...

В этой главе

- ✓ Тензоры как основная структура данных PyTorch.
- ✓ Доступ по индексу и операции над тензорами.
- ✓ Работа с многомерными массивами NumPy.
- ✓ Перенос вычислений на GPU для ускорения вычислений.

В предыдущей главе мы прошли по некоторым из множества приложений глубокого обучения. Они неизменно включали в себя получение данных в какой-либо форме, например в форме изображений или текста, и формирование данных для вывода в другой форме, например в виде меток, числовых значений или новых изображений или текста. С этой точки зрения глубокое обучение представляет собой создание системы, преобразующей данные из одного представления в другое. Подобное преобразование основывается на обнаружении каких-либо общих черт из ряда примеров данных, отражающих желаемое соответствие. Например, система может замечать общие очертания собаки и типичные для золотистого ретривера цвета. Путем сочетания этих двух свойств изображения система правильно сопоставляет изображения с заданными очертаниями и цветами и метку золотистого ретривера, а не черного лабрадора (или кошки песочного цвета, например). Получаемая в результате система может потреблять большие наборы однотипных входных данных и выдавать для них осмысленный выходной сигнал.

Процесс начинается с преобразования входных данных в числа с плавающей запятой. Преобразование пикселей изображений в числа, показанное на первом шаге на рис. 3.1, мы обсудим в главе 4 (а также многие другие типы данных). Но прежде чем дойти до этого, поговорим о том, как тензоры помогают работать со значениями с плавающей запятой в PyTorch.

3.1. МИР КАК ЧИСЛА С ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Поскольку нейронные сети работают с информацией с помощью чисел с плавающей запятой, нам нужен способ закодировать нужные данные реального мира во что-то понятное для сети, а затем декодировать выходной сигнал во что-то понятное нам и подходящее для наших целей.

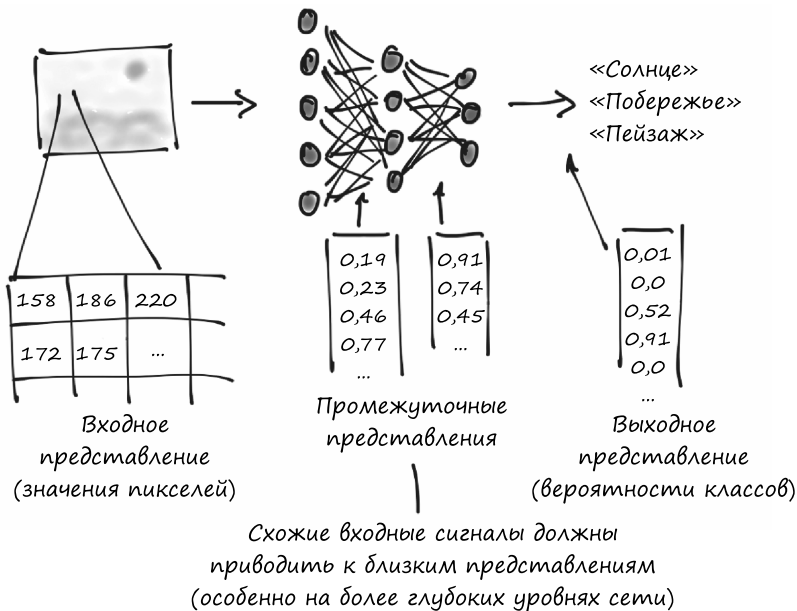


Рис. 3.1. Глубокая нейронная сеть обучается преобразовывать входное представление в выходное (примечание: указано условное количество нейронов и выходных сигналов)

Глубокие нейронные сети обычно обучаются преобразованиям одной формы данных в другую поэтапно, так что частично преобразованные данные на каждом из этапов можно считать последовательностью промежуточных представлений. В задаче распознавания изображений первые представления могут служить для захвата, например, краев изображения или определенных структур на

изображении, таких как мех. Более глубокие представления могут захватывать более сложные структуры, например глаза, носы, уши людей.

В целом, подобные промежуточные представления являются наборами чисел с плавающей запятой, которые описывают входные данные и отражают структуру входных данных способом, удобным для описания соответствий входных сигналов нейронной сети выходным. Подобные характеристики, свои для каждой задачи, усваиваются из соответствующих примеров данных. Эти наборы чисел с плавающей запятой и операции над ними — важная составляющая современного ИИ — мы встретим немало примеров этого на протяжении всей книги.

Важно не забывать, что эти промежуточные представления (например, те, что показаны на втором шаге рис. 3.1) являются результатом сочетания входных сигналов с весовыми коэффициентами предыдущего слоя нейронов. Каждое промежуточное представление уникально относительно своего предшествующего входного сигнала.

Прежде чем приступить к преобразованию данных во входные сигналы с плавающей запятой, необходимо четко разобраться, как PyTorch обрабатывает и хранит данные: входной сигнал, промежуточные представления и выходной сигнал. Именно этому и посвящена данная глава.

Для этой цели PyTorch использует базовую структуру данных: *тензор*. Мы уже сталкивались с тензорами в главе 2, когда производили вывод на основе предобученных сетей. Математики, физики и инженеры связывают тензоры с понятиями пространств, систем отсчета и преобразований между ними. Как бы то ни было, здесь об этих понятиях речь не идет. В контексте глубокого обучения тензоры связаны с обобщением векторов и матриц на произвольную размерность, как можно видеть на рис. 3.2. Говоря другими словами, речь идет о *многомерных массивах*. Размерность тензора соответствует числу индексов, при помощи которых можно ссылаться на скалярные значения внутри тензора.

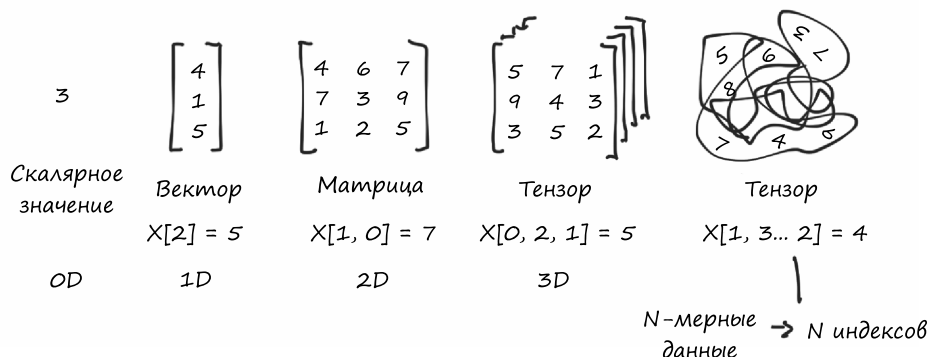


Рис. 3.2. Тензоры — базовые элементы представления данных в PyTorch

PyTorch не единственная библиотека для работы с многомерными массивами. Наиболее популярная библиотека для работы с многомерными массивами, безусловно, NumPy, которая настолько распространена, что сейчас это фактически *лингва франка* науки о данных. PyTorch обеспечивает идеальную интеграцию с прочими научными библиотеками Python, в частности SciPy (www.scipy.org), Scikit-learn (<https://scikit-learn.org>) и Pandas (<https://pandas.pydata.org>).

По сравнению с массивами NumPy тензоры PyTorch обладают несколькими потрясающими способностями, например возможностью чрезвычайно быстро выполнять операции на графических процессорах (GPU), умением распределять операции по нескольким устройствам или машинам, а также отслеживать породивший их граф вычислений. Все эти качества очень важны для реализации современной библиотеки глубокого обучения.

Мы начнем эту главу со знакомства с основами тензоров PyTorch, чтобы заложить фундамент для оставшейся части данной книги. Прежде всего мы научимся производить операции над тензорами с помощью тензорной библиотеки PyTorch, что включает в себя информацию о том, как данные хранятся в оперативной памяти, как производить определенные операции над тензорами произвольно большого размера за постоянное время, а также о вышеупомянутых возможностях взаимодействия с NumPy и GPU-ускорением. Если мы хотим сделать тензоры основным инструментом в нашем наборе, то необходимо понимать их возможности и API. В следующей главе мы применим эти знания на практике и научимся представлять различные типы данных так, как требуется для обучения нейронных сетей.

3.2. ТЕНЗОРЫ: МНОГОМЕРНЫЕ МАССИВЫ

Мы уже знаем, что тензоры — основная структура данных PyTorch. Тензор — это массив, то есть структура данных, хранящая набор чисел, к которым можно обращаться по отдельности с помощью индекса, причем используемых индексов может быть несколько.

3.2.1. От списков Python к тензорам PyTorch

Взглянем, как происходит доступ по индексу к спискам, и сравним его с доступом по индексу к тензорам. Возьмем список из трех чисел в Python (.code/p1ch3/1_tensors.ipynb):

```
# In[1]:
a = [1.0, 2.0, 1.0]
```

К элементам этого списка можно обратиться по соответствующим индексам, начиная с нулевого:

```
# In[2]:
a[0]

# Out[2]:
1.0

# In[3]:
a[2] = 3.0
a

# Out[3]:
[1.0, 2.0, 3.0]
```

Простые программы на Python, работающие с векторами числовых значений, например с координатами двумерной прямой, нередко используют для хранения этих векторов списки. Как мы увидим в следующей главе, с помощью более эффективной структуры данных — тензоров — можно представлять множество типов данных, от изображений до временных рядов. Путем описания операций с тензорами, некоторые из которых мы обсудим в этой главе, можно легко и эффективно разбивать данные на сегменты и одновременно работать со входной информацией даже на таком языке высокого уровня (и не слишком быстром), как Python.

3.2.2. Создаем наши первые тензоры

Создадим наш первый тензор PyTorch и посмотрим, что он собой представляет. Пока этот тензор не будет нести никакого особого смысла, просто три единицы в одном столбце:

```
# In[4]:
import torch  ← Импортируем модуль torch
a = torch.ones(3)  ← Создаем одномерный тензор размером 3, заполненный единицами
a

# Out[4]:
tensor([1., 1., 1.])

# In[5]:
a[1]

# Out[5]:
tensor(1.)

# In[6]:
float(a[1])

# Out[6]:
1.0

# In[7]:
```



```
a[2] = 2.0
a

# Out[7]:
tensor([1., 1., 2.])
```

После импорта модуля `torch` вызываем функцию, создающую одномерный тензор размером 3, заполненный значениями `1.0`. Обращаться к элементам можно по индексу, начиная с 0, либо путем присвоения ему нового значения. Хотя внешне этот пример не слишком отличается от списка числовых объектов, «за кулисами» все сильно отличается.

3.2.3. Что такое тензоры

Списки и кортежи числовых значений Python представляют собой наборы объектов Python, память под которые выделяется по отдельности, как показано слева на рис. 3.3. Тензоры PyTorch и массивы NumPy, с другой стороны, являются представлениями над (обычно) непрерывными блоками памяти, содержащими *распакованные* (*unboxed*) числовые типы данных C, а не объекты Python. В данном случае каждый элемент представляет собой 32-битное (4 байта) значение типа `float`, как видно на рис. 3.3, *справа*. А это значит, что для хранения 1 000 000 чисел типа `float` потребуется непрерывная область памяти в 4 000 000 байт плюс небольшое дополнительное место для метаданных (размерность и числовой тип).

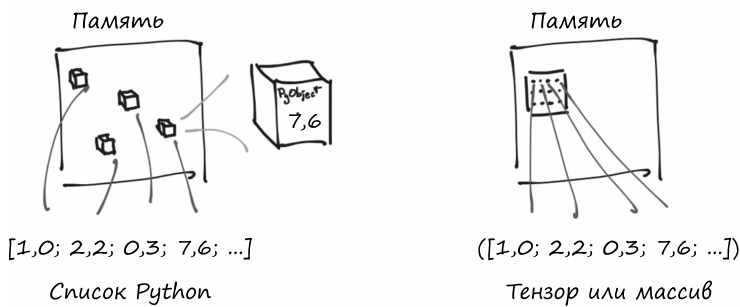


Рис. 3.3. (Упакованные) числовые значения объекта Python по сравнению с числовыми значениями тензора (распакованного массива)

Пусть дан список координат, с помощью которого мы хотим представить геометрическую фигуру: например, двумерный треугольник с вершинами в точках (4, 1), (5, 3) и (2, 1). Этот пример не относится к сфере глубокого обучения, но он простой и наглядный. Вместо хранения координат в виде числовых значений

84 Часть I. Основы PyTorch

в списке Python, как мы делали ранее, можно воспользоваться одномерным тензором и хранить x -координаты по четным индексам и y -координаты — по нечетным, вот так:

```
# In[8]:
points = torch.zeros(6)  ← С помощью .zeros мы просто получаем массив нужного размера
points[0] = 4.0          ← Записываем поверх этих нулей нужные нам значения
points[1] = 1.0
points[2] = 5.0
points[3] = 3.0
points[4] = 2.0
points[5] = 1.0
```

Можно также передать список Python конструктору и получить тот же результат:

```
# In[9]:
points = torch.tensor([4.0, 1.0, 5.0, 3.0, 2.0, 1.0])
points

# Out[9]:
tensor([4., 1., 5., 3., 2., 1.])
```

Для получения координат первой точки делаем следующее:

```
# In[10]:
float(points[0]), float(points[1])

# Out[10]:
(4.0, 1.0)
```

Вполне допустимый вариант, хотя нам было бы удобнее, чтобы первый индекс ссылался на отдельные двумерные точки, а не координаты точек. Для этой цели можно воспользоваться двумерным тензором:

```
# In[11]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[11]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

Здесь мы передали в конструктор список списков. Запрашиваем форму тензора:

```
# In[12]:
points.shape

# Out[12]:
torch.Size([3, 2])
```

Мы получили информацию о размере тензора по каждому из измерений. Можно также воспользоваться функциями `zeros` или `ones` для задания начальных значений тензора, для чего необходимо передать в них его размер в виде кортежа:

```
# In[13]:
points = torch.zeros(3, 2)
points
```

```
# Out[13]:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

Теперь можно обращаться к отдельным элементам тензора, указав два индекса:

```
# In[14]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points
```

```
# Out[14]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

```
# In[15]:
points[0, 1]
```

```
# Out[15]:
tensor(1.)
```

Этот оператор возвращает *y*-координату нулевой точки нашего набора данных. Можно также обращаться к первому элементу тензора так, как мы делали ранее, для получения двумерных координат первой точки:

```
# In[16]:
points[0]
```

```
# Out[16]:
tensor([4., 1.])
```

На выходе получается еще один тензор, с другим *представлением* (*view*) тех же самых данных. Новый тензор — одномерный, размером 2, и использующий значения из первой строки тензора `points`. Значит ли это, что был выделен новый участок памяти, в который были скопированы значения, и новая память была возвращена завернутой в новый объект тензора? Нет, поскольку такая реализация была бы очень неэффективной, особенно в случае миллионов точек. Мы позже еще вернемся к вопросу хранения тензоров в этой главе, когда будем обсуждать представления тензоров в разделе 3.7.

3.3. ДОСТУП К ТЕНЗОРАМ ПО ИНДЕКСАМ

Что, если нужно получить тензор, содержащий все точки данных, кроме первой? Это легко сделать с помощью нотации диапазонного доступа по индексу, применимой также к обычным спискам Python. Вспомним, как это делается:

```

# In[53]:
some_list = list(range(6))
some_list[:]
some_list[1:4]
some_list[1:]
some_list[:4]
some_list[:-1]
some_list[1:4:2]

```

Все элементы списка

С 1-го элемента включительно по 4-й элемент, не включая его

С 1-го элемента включительно до конца списка

С начала списка по 4-й элемент, не включая его

С начала списка по предпоследний элемент

С 1-го элемента включительно по 4-й элемент, не включая его, через один элемент

Для достижения наших целей можно использовать ту же самую нотацию применительно к тензорам PyTorch с дополнительным плюсом в виде возможности использовать диапазонный доступ по индексу для каждого из измерений тензора:

```

# In[54]:
points[1:]
points[1:, :]
points[1:, 0]
points[None]

```

Все строки после первой; все столбцы (неявно)

Все строки после первой; все столбцы

Все строки после первой; первый столбец

Добавляет измерение размера 1, подобно функции `unsqueeze`

Помимо диапазонов, PyTorch может похвастаться обладающей большими возможностями формой доступа по индексу — *расширенным доступом по индексу* (*advanced indexing*), о котором мы поговорим в следующей главе.

3.4. ПОИМЕНОВАННЫЕ ТЕНЗОРЫ

Измерения (оси координат) тензоров обычно соответствуют чему-то наподобие расположения пикселей или цветовых каналов. А это значит, что при доступе к тензору по индексам необходимо помнить порядок измерений и записывать индексы соответствующим образом. Отслеживание того, какое измерение содержит какие данные, может приводить к возникновению ошибок при преобразовании данных через несколько тензоров.

Приведем конкретный пример. Пусть дан трехмерный тензор, наподобие `img_t` из подраздела 2.1.4 (для простоты мы воспользуемся тут фиктивными данными), который необходимо преобразовать в оттенки серого. Ищем типичные весовые коэффициенты цветов для получения одного единого значения яркости¹:

```
# In[2]:
img_t = torch.randn(3, 5, 5) # форма [каналы, строки, столбцы]
weights = torch.tensor([0.2126, 0.7152, 0.0722])
```

От кода также нередко требуются возможности обобщения: например, изображений в оттенках серого, представленных в виде двумерных тензоров с измерениями высоты и ширины, до цветных изображений с третьим цветовым каналом (как в RGB), или одного изображения до пакета изображений. В подразделе 2.1.4 мы ввели дополнительное измерение батчей в `batch_t`, здесь же мы будем считать, что размер батча равен 2:

```
# In[3]:
batch_t = torch.randn(2, 3, 5, 5) # форма [батч, каналы, строки, столбцы]
```

Иногда каналы RGB размещаются в измерении 0, а иногда — в измерении 1. Но обобщение можно производить путем отсчета с конца: они всегда расположены в измерении `-3`, третьем с конца. Вычисление невзвешенного среднего можно записать следующим образом:

```
# In[4]:
img_gray_naive = img_t.mean(-3)
batch_gray_naive = batch_t.mean(-3)
img_gray_naive.shape, batch_gray_naive.shape
```

```
# Out[4]:
(torch.Size([5, 5]), torch.Size([2, 5, 5]))
```

Но теперь у нас есть и веса. PyTorch позволяет перемножать объекты одинаковой формы, либо когда один из них имеет размер 1 по заданному измерению. Кроме того, PyTorch автоматически добавляет в начало измерения размером 1. Эта функция называется *транслированием* (*broadcasting*). `batch_t` формы (2, 3, 5, 5) умножается на `unsqueezed_weights` формы (3, 1, 1), в результате чего получается тензор формы (2, 3, 5, 5), в котором затем можно сложить третье измерение с конца (три канала):

```
# In[5]:
unsqueezed_weights = weights.unsqueeze(-1).unsqueeze_(-1)
img_weights = (img_t * unsqueezed_weights)
batch_weights = (batch_t * unsqueezed_weights)
```

¹ Поскольку стандартизировать субъективное восприятие не так-то просто, было выбрано немало вариантов весовых коэффициентов. Например, см. [https://en.wikipedia.org/wiki/Luma_\(video\)](https://en.wikipedia.org/wiki/Luma_(video)).

```
img_gray_weighted = img_weights.sum(-3)
batch_gray_weighted = batch_weights.sum(-3)
batch_weights.shape, batch_t.shape, unsqueezed_weights.shape
```

```
# Out[5]:
(torch.Size([2, 3, 5, 5]), torch.Size([2, 3, 5, 5]), torch.Size([3, 1, 1]))
```

Поскольку такие операции быстро становятся запутанными, а также ради повышения эффективности — функция `einsum` (адаптированная из NumPy) включает мини-язык¹ описания доступа по индексам, позволяющий давать названия индексам измерений для суммирования подобных произведений. Как это часто бывает в Python, транслирование — форма приведения неименованных объектов к одной форме — выполняется с помощью трех точек '...'; но не беспокойтесь по поводу `einsum` слишком сильно, потому что она нам больше не понадобится:

```
# In[6]:
img_gray_weighted_fancy = torch.einsum('...chw,c->...hw', img_t, weights)
batch_gray_weighted_fancy = torch.einsum('...chw,c->...hw', batch_t, weights)
batch_gray_weighted_fancy.shape

# Out[6]:
torch.Size([2, 5, 5])
```

Как видим, количество вспомогательных операций здесь довольно велико, что чревато ошибками, особенно если места создания и использования тензоров находятся далеко друг от друга в нашем коде. Это обстоятельство привлекло внимание специалистов-практиков, поэтому было предложено² давать измерениям названия.

В PyTorch 1.3 добавилась экспериментальная возможность *именования тензоров* (см. https://pytorch.org/tutorials/intermediate/named_tensor_tutorial.html и https://pytorch.org/docs/stable/named_tensor.html). У функций создания тензоров, например `tensor` и `rand`, есть аргумент `names`. В качестве аргумента `names` должна передаваться последовательность строковых значений:

```
# In[7]:
weights_named = torch.tensor([0.2126, 0.7152, 0.0722], names=['channels'])
weights_named

# Out[7]:
tensor([0.2126, 0.7152, 0.0722], names=('channels',))
```

При необходимости добавить названия в имеющийся тензор (не меняя существующие) можно вызвать его метод `refine_names`. Аналогично доступу по

¹ Неплохой его обзор можно найти в сообщении из блога Тима Роктэшеля (Tim Rocktäschel) *Einsum is All You Need — Einstein Summation in Deep Learning* (<https://rockt.github.io/2018/04/30/einsum>).

² См.: Sasha Rush, *Tensor Considered Harmful*, Harvardnlp, <http://nlp.seas.harvard.edu/NamedTensor>.

индексу с помощью многоточия можно пропускать любое количество измерений. С помощью родственного ему метода `rename` можно также переопределять или удалять (путем передачи `None`) уже существующие названия:

```
# In[8]:
img_named = img_t.refine_names(..., 'channels', 'rows', 'columns')
batch_named = batch_t.refine_names(..., 'channels', 'rows', 'columns')
print("img named:", img_named.shape, img_named.names)
print("batch named:", batch_named.shape, batch_named.names)

# Out[8]:
img named: torch.Size([3, 5, 5]) ('channels', 'rows', 'columns')
batch named: torch.Size([2, 3, 5, 5]) (None, 'channels', 'rows', 'columns')
```

Для операций с двумя входами, помимо обычных проверок измерений, таких как сравнение размеров, или если размер одного равен 1 и его можно транслировать на другой, PyTorch во всех случаях будет проверять имена. PyTorch еще не выравнивает измерения автоматически, так что необходимо делать это явным образом. Метод `align_as` возвращает тензор, в котором добавлены недостающие измерения, а уже существующие переставлены в нужном порядке:

```
# In[9]:
weights_aligned = weights_named.align_as(img_named)
weights_aligned.shape, weights_aligned.names

# Out[9]:
(torch.Size([3, 1, 1]), ('channels', 'rows', 'columns'))
```

Функции, принимающие на входе аргументы для измерений, также позволяют указывать поименованные измерения:

```
# In[10]:
gray_named = (img_named * weights_aligned).sum('channels')
gray_named.shape, gray_named.names

# Out[10]:
(torch.Size([5, 5]), ('rows', 'columns'))
```

При попытке сочетать измерения с различными названиями выдается сообщение об ошибке:

```
gray_named = (img_named[..., :3] * weights_named).sum('channels')

RuntimeError: Error when
  attempting to broadcast dims ['channels', 'rows',
    'columns'] and dims ['channels']: dim 'columns' and dim 'channels'
  are at the same position from the right but do not match.
```

При необходимости использовать тензоры не только в функциях, работающих с поименованными тензорами, необходимо удалить названия, установив их в `None`. Следующий код возвращает нас в мир безымянных измерений:

```
# In[12]:
gray_plain = gray_named.rename(None)
gray_plain.shape, gray_plain.names

# Out[12]:
(torch.Size([5, 5]), (None, None))
```

Учитывая экспериментальный статус этой возможности на момент написания данной книги и чтобы не возиться с доступом по индексу и выравниванием, мы будем использовать безымянные тензоры в оставшейся части данной книги. Поименованные тензоры позволяют исключить многие источники ошибок выравнивания, которые, если верить форуму PyTorch, могут доставить немало головной боли. Интересно будет взглянуть, насколько широко они станут применяться в будущем.

3.5. ТИПЫ ЭЛЕМЕНТОВ ТЕНЗОРОВ

Пока мы охватили основы работы тензоров, но не обсуждали, какие виды числовых типов данных можно хранить в `Tensor`. Как мы говорили в разделе 3.2, использовать стандартные типы данных Python не рекомендуется по нескольким причинам.

- *Числовые значения в Python являются объектами.* В то время как число с плавающей запятой требует для представления в компьютере только 32 бита, Python преобразует его в полноценный объект Python с подсчетом ссылок и т. д. Эта операция, которая называется *упаковкой (boxing)*, не является проблемой при хранении небольшого количества числовых значений, но выделять память для миллионов таких объектов — совершенно нерационально.
- *Списки в Python предназначены для хранения последовательных наборов объектов.* В них нет операций для быстрого вычисления скалярного произведения двух векторов или их суммирования. Кроме того, списки Python не оптимизируют размещение своего содержимого в памяти, поскольку представляют собой наборы указателей на объекты Python (любые, не только числовые значения) с доступом по индексу. Наконец, списки Python одномерны, и, хотя можно создавать списки списков, это тоже нерационально.
- *Интерпретатор Python работает медленно по сравнению с оптимизированным, скомпилированным кодом.* Написанный на низкоуровневом языке, таком как C, оптимизированный код может производить математические операции над большими наборами числовых данных намного быстрее.

Поэтому библиотеки для исследования данных используют NumPy или вводят специализированные структуры данных наподобие тензоров PyTorch, обеспечивающие эффективные низкоуровневые реализации числовых структур

данных и соответствующих операций над ними, с предоставлением удобного высокоуровневого API. Для этого все объекты в тензоре должны быть числовыми значениями одинакового типа, а PyTorch должен этот тип понимать.

3.5.1. Задание числового типа с помощью dtype

Аргумент `dtype` конструкторов тензоров (то есть таких функций, как `tensor`, `zeros` и `ones`) позволяет задавать тип содержащихся в тензоре числовых данных. Тип данных задает спектр значений, которые могут храниться в тензоре (целые числа или числа с плавающей запятой), и количество байтов, занимаемое каждым значением¹. Аргумент `dtype` специально был сделан идентичным стандартному аргументу NumPy с тем же названием. Вот список возможных значений аргумента `dtype`:

- `torch.float32 (torch.float)`: 32-битное значение с плавающей запятой;
- `torch.float64 (torch.double)`: 64-битное значение с плавающей запятой, с двойной точностью;
- `torch.float16 (torch.half)`: 16-битное значение с плавающей запятой, с половинной точностью;
- `torch.int8`: знаковое 8-битное целочисленное значение;
- `torch.uint8`: беззнаковое 8-битное целочисленное значение;
- `torch.int16 (torch.short)`: знаковое 16-битное целочисленное значение;
- `torch.int32 (torch.int)`: знаковое 32-битное целочисленное значение;
- `torch.int64 (torch.long)`: знаковое 64-битное целочисленное значение;
- `torch.bool`: булево значение.

Тип данных по умолчанию для тензоров — это 32-битное значение с плавающей запятой.

3.5.2. dtype на все случаи жизни

Как мы увидим в следующих главах, вычисления в нейронных сетях обычно производятся над 32-битными значениями с плавающей запятой. Более высокая точность, например 64-битные значения, обычно не повышает безошибочность модели, но требует больше памяти и вычислительного времени. Нативная поддержка типа данных с половинной точностью — 16-битных значений с плавающей запятой — в стандартных CPU обычно отсутствует, зато предоставляется

¹ А также наличие знака в случае `uint8`.

современными GPU. При необходимости можно перейти на половинную точность для снижения объема занимаемой памяти нейросетевой модели без особого влияния на степень безошибочности.

Тензоры можно использовать в качестве индексов для других тензоров. В этом случае PyTorch предполагает, что тип данных тензоров-индексов будет 64-битный целочисленный. По умолчанию при целочисленных аргументах создаваемого тензора, например, как в `torch.tensor([2, 2])`, будет создан тензор с 64-битными целочисленными значениями. В результате чаще всего мы будем сталкиваться с типами данных `float32` и `int64`.

Наконец, предикаты с участием тензоров, наподобие `points > 1.0`, возвращают тензоры типа `bool`, указывающие, удовлетворяет ли условию каждый отдельный элемент. В двух словах, это и есть числовые типы.

3.5.3. Работа с атрибутом `dtype` тензоров

Для выделения памяти под тензор с нужным числовым типом данных можно указать соответствующий `dtype` в качестве аргумента конструктора. Например:

```
# In[47]:
double_points = torch.ones(10, 2, dtype=torch.double)
short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)
```

Получить информацию о `dtype` тензора можно путем обращения к соответствующему атрибуту:

```
# In[48]:
short_points.dtype

# Out[48]:
torch.int16
```

Можно также привести результат функции создания тензора к нужному типу с помощью соответствующего метода приведения типов, например:

```
# In[49]:
double_points = torch.zeros(10, 2).double()
short_points = torch.ones(10, 2).short()
```

или более удобного метода `to`:

```
# In[50]:
double_points = torch.zeros(10, 2).to(torch.double)
short_points = torch.ones(10, 2).to(dtype=torch.short)
```

«За кулисами» `to` проверяет необходимость преобразования и производит его, если нужно. Методы приведения типов `dtype`, например `float`, являются

сокращенными формами вызова метода `to`, но они могут также принимать дополнительные аргументы, которые мы обсудим в разделе 3.9.

При смешивании входных типов данных в операциях данные автоматически преобразуются к большему типу. Следовательно, если нам нужны 32-битные вычисления, нужно убедиться, что все входные сигналы (как минимум) 32-битные:

```
# In[51]:
points_64 = torch.rand(5, dtype=torch.double)
points_short = points_64.to(torch.short)
points_64 * points_short # работает, начиная с PyTorch 1.3

# Out[51]:
tensor([0., 0., 0., 0., 0.], dtype=torch.float64)
```

С помощью `rand` мы задаем случайные начальные значения элементов тензора в диапазоне от 0 до 1

3.6. API ТЕНЗОРОВ

Мы уже знаем, что представляют собой тензоры PyTorch и что происходит «за кулисами». В завершение стоит обсудить предоставляемые PyTorch операции над тензорами. Перечислять их все тут смысла не имеет. Вместо этого мы дадим вам общее представление об их API и укажем, где искать информацию в онлайн-документации, размещенной по адресу <http://pytorch.org/docs>.

Прежде всего абсолютное большинство операций над и между тензорами доступны в модуле `torch`, а также их можно вызывать как методы объекта-тензора. Например, можно использовать уже встречавшуюся нам функцию `transpose` из модуля `torch`:

```
# In[71]:
a = torch.ones(3, 2)
a_t = torch.transpose(a, 0, 1)

a.shape, a_t.shape

# Out[71]:
(torch.Size([3, 2]), torch.Size([2, 3]))

или как метод тензора:

# In[72]:
a = torch.ones(3, 2)
a_t = a.transpose(0, 1)

a.shape, a_t.shape

# Out[72]:
(torch.Size([3, 2]), torch.Size([2, 3]))
```

Никакого различия между этими двумя формами вызова нет, они равноценны.

Мы уже упоминали выше онлайн-документацию (<http://pytorch.org/docs>). Она исчерпывающая и отлично организованная, а операции над тензорами разбиты по группам.

- *Операции создания* — функции для формирования тензоров, например `ones` и `from_numpy`.
- *Операции доступа по индексу, нарезки, объединения и перестановки элементов* — функции изменения формы, шага или содержимого тензоров, например `transpose`.
- *Математические операции* — функции для вычислительных операций над содержимым тензоров.
- *Поэлементные операции* — функции получения нового тензора посредством применения функции к каждому из элементов исходного тензора по отдельности, например `abs` и `cos`.
- *Операции свертки* — функции вычисления сводных показателей посредством прохода по тензорам в цикле, например `mean`, `std` и `norm`.
- *Операции сравнения* — функции вычисления числовых предикатов над тензорами, например `equal` и `max`.
- *Спектральные операции* — функции для преобразований и операций в частотном диапазоне, например `stft` и `hamming`.
- *Прочие операции* — специальные функции для работы с векторами, например `cross`, или матрицами, например `trace`.
- *Операции BLAS и LAPACK* — функции, следующие спецификациям BLAS (Basic Linear Algebra Subprograms, «основные подпрограммы линейной алгебры») по операциям над скалярными значениями, а также операциям с двумя операндами типа «вектор — вектор», «матрица — вектор», «матрица — матрица».
- *Случайные выборки* — функции для генерации значений путем случайной выборки из распределений вероятности, например `randn` и `normal`.
- *Сериализация* — функции для сохранения и загрузки тензоров, например `load` и `save`.
- *Распараллеливание* — функции для управления количеством потоков выполнения для параллельного выполнения на CPU, например `set_num_threads`.

Потратьте немного времени на эксперименты с общим API тензоров. В этой главе приведены все необходимые предварительные сведения для подобного интерактивного изучения. С некоторыми операциями над тензорами мы также встретимся по ходу этой книги, начиная со следующей главы.

3.7. ТЕНЗОРЫ: ХРАНЕНИЕ В ПАМЯТИ

Пора заглянуть «за кулисы» и внимательнее изучить особенности реализации. Память под значения в тензорах выделяется непрерывными фрагментами под управлением экземпляров `torch.Storage`. Хранилище представляет собой одномерный массив числовых данных, то есть непрерывный фрагмент памяти, содержащий числа заданного типа, например `float` (32-битные значения, выражающие числа с плавающей запятой) или `int64` (64-битные значения, выражающие целые числа). Экземпляр класса `Tensor` PyTorch — это представление подобного экземпляра `Storage` с возможностью доступа к хранилищу по индексу через указание сдвига и шага по каждому измерению¹.

Несколько тензоров могут обращаться по индексам к одному хранилищу, даже если индексация данных происходит по-разному. Мы рассмотрим пример этого на рис. 3.4. На самом деле, когда мы запрашивали `points[0]` в разделе 3.2, мы получали еще один тензор, индексирующий то же хранилище, что и тензор `points`, — только не все и другой размерности (одномерное, а не двумерное). Впрочем, лежащая в его основе память выделяется только один раз, благодаря чему создание различных тензорных представлений данных происходит очень быстро вне зависимости от размера данных, контролируемых экземпляром `Storage`.

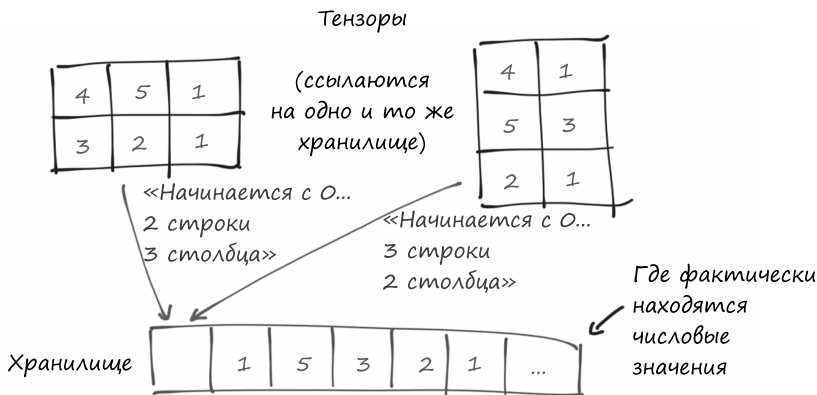


Рис. 3.4. Тензоры — это представления экземпляров `Storage`

3.7.1. Доступ к хранилищу по индексу

Давайте взглянем, как происходит доступ к хранилищу по индексу на практике, на примере наших двумерных точек. Обращаться к хранилищу конкретного тензора можно посредством его свойства `.storage`:

¹ В будущих выпусках PyTorch, возможно, прямой доступ к `Storage` будет убран, но то, что вы здесь увидите, все равно неплохо отражает внутренние механизмы работы тензоров.

```
# In[17]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points.storage()

# Out[17]:
4.0
1.0
5.0
3.0
2.0
1.0
[torch.FloatTensor of size 6]
```

И хотя тензор сообщает, что содержит три строки и два столбца, лежащее в его основе хранилище представляет собой непрерывный массив размером 6. В этом смысле тензор просто знает, как преобразовать пару индексов в место в этом хранилище.

Можно также обращаться к хранилищу по индексу вручную. Например:

```
# In[18]:
points_storage = points.storage()
points_storage[0]

# Out[18]:
4.0

# In[19]:
points.storage()[1]

# Out[19]:
1.0
```

Мы не можем обращаться по индексу к хранилищу двумерного тензора с помощью двух индексов. Хранилище всегда представляет собой одномерный массив вне зависимости от размерности каких-либо ссылающихся на него тензоров.

Теперь вас вряд ли удивит, что изменение значения в хранилище приводит к изменению содержимого тензора, который на него ссылается:

```
# In[20]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_storage = points.storage()
points_storage[0] = 2.0
points

# Out[20]:
tensor([[2., 1.],
        [5., 3.],
        [2., 1.]])
```

3.7.2. Модификация хранимых значений: операции с заменой на месте

Помимо представленных в предыдущем разделе операций над тензорами, небольшое количество операций доступно только в виде методов объекта `Tensor`. Их можно отличить по подчеркиванию в конце названия, как в `zero_`, указывающему, что метод работает *с заменой на месте* (*in place*), изменяя входные данные вместо того, чтобы создавать новый выходной тензор и возвращать его. Например, метод `zero_` обнуляет все элементы входного тензора. Все методы, в конце названия которых нет символа подчеркивания, оставляют исходный тензор неизменным и вместо этого возвращают новый:

```
# In[73]:
a = torch.ones(3, 2)

# In[74]:
a.zero_()
a

# Out[74]:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

3.8. МЕТАДААННЫЕ ТЕНЗОРОВ: РАЗМЕР, СДВИГ И ШАГ

Для доступа к хранилищу по индексу в тензорах используется несколько элементов информации, которые вместе с хранилищем однозначно определяют их: размер, сдвиг и шаг. Схема их взаимосвязи приведена на рис. 3.5. Размер (форма в NumPy) представляет собой кортеж, содержащий число элементов тензора по каждому измерению. Сдвиг хранилища — это индекс в хранилище, соответствующий первому элементу тензора. Шаг — это количество элементов хранилища, пропускаемых между последовательными элементами по каждому измерению.

3.8.1. Представления хранилища другого тензора

Мы можем получить вторую точку из тензора, указав соответствующий индекс:

```
# In[21]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1]
second_point.storage_offset()

# Out[21]:
2
```

```
# In[22]:
second_point.size()
```

```
# Out[22]:
torch.Size([2])
```

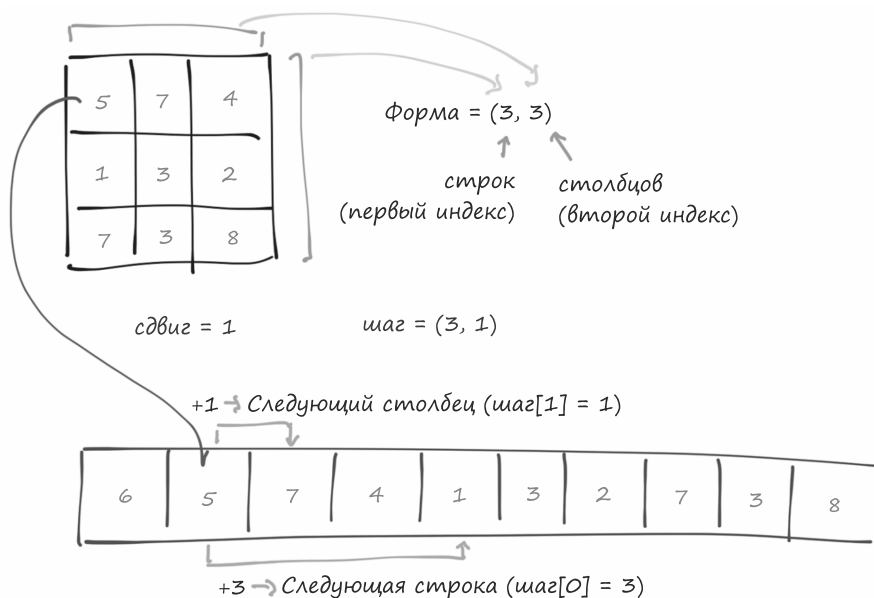


Рис. 3.5. Взаимосвязь между размером, сдвигом и шагом тензора. В данном случае тензор является представлением большого хранилища, например выделенного при создании большого тензора

Сдвиг полученного в результате тензора в хранилище равен 2 (поскольку мы пропускаем первую точку, содержащую два элемента), а размер представляет собой экземпляр класса `Size`, содержащий один элемент, поскольку наш тензор одномерный. Важно отметить, что это та же информация, что содержится в свойстве `shape` объектов-тензоров:

```
# In[23]:
second_point.shape
```

```
# Out[23]:
torch.Size([2])
```

Шаг представляет собой кортеж, указывающий число элементов в хранилище, пропускаемое при увеличении индекса на 1 по каждому измерению. Например, шаг нашего тензора `points` равен `(2, 1)`:


```
# In[24]:
points.stride()

# Out[24]:
(2, 1)
```

Обращение к элементу i, j двумерного тензора означает обращение к элементу $\text{сдвиг_хранилища} + \text{шаг}[0] * i + \text{шаг}[1] * j$ хранилища. Сдвиг обычно равен 0, но если данный тензор является представлением хранилища, созданного для хранения большего тензора, он может быть больше нуля. Подобное преобразование индексов между объектами `Tensor` и `Storage` снижает затраты на выполнение некоторых операций, например транспонирование тензора или выделение из него подтензора, поскольку не требуется перераспределять память. Вместо этого память выделяется под новый объект `Tensor` с другими значениями размера, сдвига и шага.

Нам уже приходилось выделять подтензор, когда мы извлекали по индексу конкретную точку и наблюдали увеличение сдвига хранилища. Давайте взглянем, что происходит при этом с размером и шагом:

```
# In[25]:
second_point = points[1]
second_point.size()

# Out[25]:
torch.Size([2])

# In[26]:
second_point.storage_offset()

# Out[26]:
2

# In[27]:
second_point.stride()

# Out[27]:
(1,)
```

Нижняя строка указывает, что размерность подтензора на единицу меньше, как и можно было ожидать, но доступ по индексу производится к тому же хранилищу, что и у исходного тензора `points`. А это значит, что побочным эффектом изменения подтензора станет изменение исходного тензора:

```
# In[28]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1]
second_point[0] = 10.0
points

# Out[28]:
tensor([[ 4.,  1.],
        [10.,  3.],
        [ 2.,  1.]])
```

Такое поведение не всегда желательно, так что имеет смысл перезаписать наш подтензор в новый тензор:

```
# In[29]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1].clone()
second_point[0] = 10.0
points

# Out[29]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

3.8.2. Транспонирование без копирования

Попробуем теперь транспонировать тензор. Возьмем наш тензор `points`, в котором отдельные точки отсчитываются по строкам, а координаты x и y — по столбцам, и транспонируем его, чтобы отдельные точки отсчитывались по столбцам. Воспользуемся этим случаем, чтобы познакомить вас с функцией `t` — сокращенной записью функции `transpose` для двумерных тензоров:

```
# In[30]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[30]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])

# In[31]:
points_t = points.t()
points_t

# Out[31]:
tensor([[4., 5., 2.],
        [1., 3., 1.]])
```

СОВЕТ

Чтобы лучше разобраться во внутреннем устройстве тензоров, рекомендуем взять карандаш и бумагу и рисовать схемы, подобные изображенной на рис. 3.5, по мере чтения кода в этом разделе.

Можно легко убедиться, что хранилище у этих двух тензоров одно:

```
# In[32]:
id(points.storage()) == id(points_t.storage())

# Out[32]:
True
```

и что они отличаются только формой и шагом:

```
# In[33]:
points.stride()

# Out[33]:
(2, 1)
# In[34]:
points_t.stride()

# Out[34]:
(1, 2)
```

Отсюда ясно, что при увеличении первого индекса в `points` на единицу (например, с `points[0,0]` до `points[1,0]`) в хранилище пропускаются два элемента, а при увеличении второго — с `points[0,0]` до `points[0,1]` — пропускается один. Другими словами, элементы тензора хранятся в этом хранилище построчно.

Можно транспонировать `points` в `points_t`, как показано на рис. 3.6. Мы меняем порядок элементов в шаге. Если затем увеличить номер строки (первый индекс тензора), в хранилище будет пропущен один элемент, точно так же, как когда мы двигались по столбцам в `points`. Это и есть транспонирование, по определению. Никакой дополнительной памяти не выделяется: транспонирование достигается путем создания нового экземпляра `Tensor` с другим порядком шагов, чем в исходном.

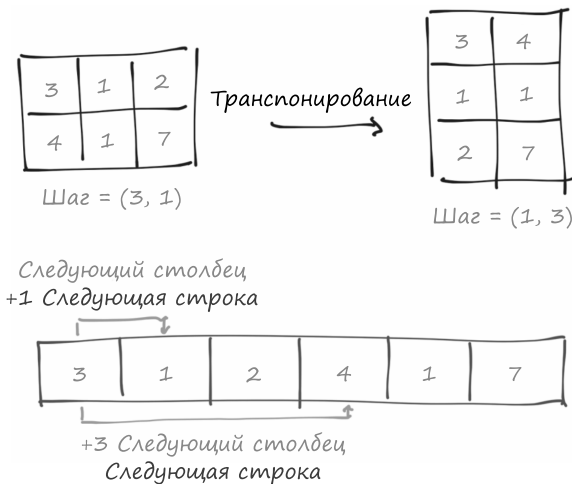


Рис. 3.6. Применение к тензору операции транспонирования

3.8.3. Транспонирование при более высокой размерности

Транспонировать в PyTorch можно не только матрицы. Можно транспонировать многомерный массив, и для этого достаточно указать два измерения, по которым нужно произвести транспонирование (зеркально отражая форму и шаг):

```

# In[35]:
some_t = torch.ones(3, 4, 5)
transpose_t = some_t.transpose(0, 2)
some_t.shape

# Out[35]:
torch.Size([3, 4, 5])

# In[36]:
transpose_t.shape

# Out[36]:
torch.Size([5, 4, 3])

# In[37]:
some_t.stride()

# Out[37]:
(20, 5, 1)

# In[38]:
transpose_t.stride()

# Out[38]:
(1, 5, 20)

```

Тензоры, значения которых размещаются в хранилище, начиная с крайнего справа измерения и далее (то есть по строкам для двумерного тензора), определяются как *непрерывные* (*contiguous*). Непрерывные тензоры удобны тем, что их можно эффективно просматривать по порядку, не прыгая по хранилищу от одного элемента к другому (улучшение локальности данных повышает производительность благодаря тому, как происходит доступ к памяти в современных CPU). Конечно, это зависит от способа выполнения алгоритмов обращения.

3.8.4. Непрерывные тензоры

Некоторые тензорные операции наподобие `view`, с которой мы встретимся в следующей главе, в PyTorch работают только для непрерывных тензоров. В подобном случае PyTorch формирует информативное исключение и требует явного вызова функции `contiguous`. Стоит отметить, что если тензор уже непрерывный, то никаких действий при вызове `contiguous` не производится (и на производительности это никак не сказывается).

В нашем случае тензор `points` непрерывный, а транспонированный к нему — нет:

```

# In[39]:
points.is_contiguous()

# Out[39]:
True

# In[40]:

```

```
points_t.is_contiguous()
```

```
# Out[40]:
False
```

Получить новый непрерывный тензор из ненепрерывного можно с помощью метода `contiguous`. Содержимое этого нового тензора будет таким же, но шаг, как и хранилище, изменится:

```
# In[41]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_t = points.t()
points_t
```

```
# Out[41]:
tensor([[4., 5., 2.],
        [1., 3., 1.]])
```

```
# In[42]:
points_t.storage()
```

```
# Out[42]:
4.0
1.0
5.0
3.0
2.0
1.0
[torch.FloatTensor of size 6]
```

```
# In[43]:
points_t.stride()
```

```
# Out[43]:
(1, 2)
```

```
# In[44]:
points_t_cont = points_t.contiguous()
points_t_cont
```

```
# Out[44]:
tensor([[4., 5., 2.],
        [1., 3., 1.]])
```

```
# In[45]:
points_t_cont.stride()
```

```
# Out[45]:
(3, 1)
```

```
# In[46]:
points_t_cont.storage()
```

```
# Out[46]:
4.0
5.0
```

```

2.0
1.0
3.0
1.0
[torch.FloatTensor of size 6]

```

Обратите внимание, что хранилище было перетасовано, чтобы в новом хранилище элементы располагались построчно. Шаг также был изменен в соответствии с новым размещением элементов.

В качестве напоминания снова покажем на рис. 3.7 нашу схему. Надеемся, она стала понятнее, когда мы изучили, как устроены тензоры.

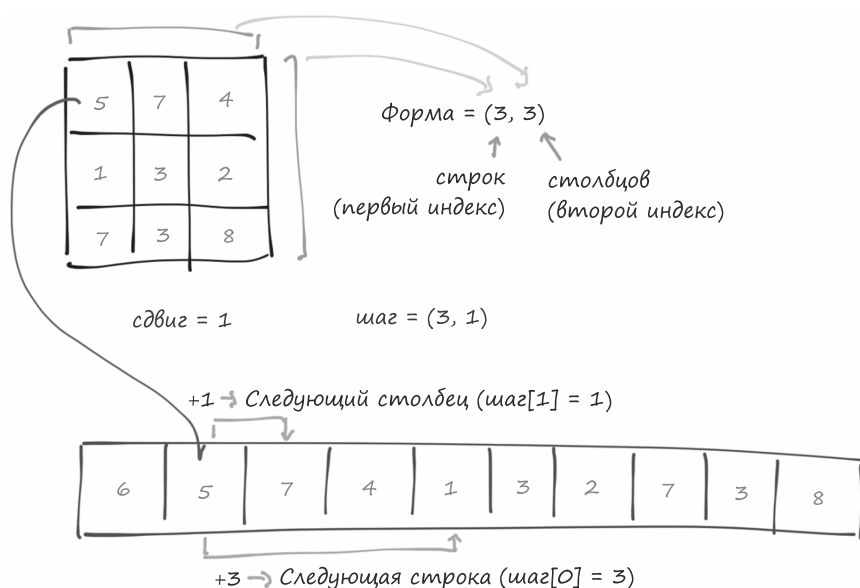


Рис. 3.7. Взаимосвязь между размером, сдвигом и шагом тензора. В данном случае тензор является представлением большего хранилища, например выделенного при создании большего тензора

3.9. ПЕРЕНОС ТЕНЗОРОВ НА GPU

До сих пор в этой главе, когда мы говорили о хранилищах, мы подразумевали память CPU. Тензоры PyTorch можно также хранить на других процессорах: графических (GPU). Любой из тензоров PyTorch можно перенести на (один из) GPU системы для массово-параллельных быстрых вычислений. После этого все производимые с этим тензором операции будут выполняться с помощью специализированных процедур для работы с GPU, включенных в PyTorch.

ПОДДЕРЖКА PYTORCH РАЗЛИЧНЫХ GPU

По состоянию на середину 2019 года основные выпуски PyTorch включали ускорение вычислений только с помощью поддерживающих CUDA GPU. PyTorch можно запустить на графическом процессоре ROCm от AMD (<https://rocm.github.io/>), и ветка master репозитория поддерживает его, но пока его необходимо компилировать самостоятельно (перед обычным процессом сборки требуется запустить сценарий `tools/amd_build/build_amd.py` для преобразования кода GPU). Понемногу продвигается (<https://github.com/pytorch/xla>) поддержка тензорных процессоров Google (TPU), текущая пробная версия доступна широкой публике в Google Colab: <https://colab.research.google.com/>. Реализация структур данных и ядер на других технологиях GPU, например OpenGL, по состоянию на момент написания данной книги не планируется.

3.9.1. Работа с атрибутом `device` тензоров

Помимо `dtype`, класс `Tensor` предоставляет атрибут `device`, который описывает, где на компьютере размещаются данные тензора. Вот как можно создать тензор в GPU, указав в конструкторе соответствующий аргумент:

```
# In[64]:
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')
```

Вместо этого можно скопировать созданный в CPU тензор на GPU с помощью метода `to`:

```
# In[65]:
points_gpu = points.to(device='cuda')
```

При этом возвращается новый тензор с теми же числовыми данными, но хранящийся в памяти GPU, а не в обычной оперативной памяти системы. Теперь, когда данные хранятся локально на GPU, начинают проявляться упомянутые выше ускорения выполнения математических операций над тензором. Практически во всех случаях тензоры на основе CPU и GPU предоставляют один и тот же API пользователя, намного упрощая написание кода, для которого не будет важно, где именно происходит вся обработка числовой информации.

Если на нашей машине более одного GPU, можно также указать, на каком именно GPU размещать тензор, передав отсчитываемый с нуля целочисленный номер GPU на машине, вот так:

```
# In[66]:
points_gpu = points.to(device='cuda:0')
```

После этого все операции над тензором, например умножение всех элементов на константу, производятся на GPU:

```
# In[67]:
points = 2 * points  ← Умножение выполняется на CPU
points_gpu = 2 * points.to(device='cuda')  ← Умножение выполняется на GPU
```

Отметим, что тензор `points_gpu` не передается обратно в CPU после вычисления результата. Вот что происходит в этой строке.

1. Тензор `points` копируется в GPU.
2. Выделяется память в GPU под новый тензор, в котором будет храниться результат умножения.
3. Возвращается обращение к этому GPU-тензору.

Следовательно, если мы прибавим к результату константу:

```
In[68]:
points_gpu = points_gpu + 4
```

операция сложения будет по-прежнему производиться в GPU и никакой информации в CPU передаваться не будет (если мы не будем выводить полученный тензор на экран или обращаться к нему). Для переноса тензора обратно в CPU необходимо указать в методе `to` аргумент `cpu`, вот так:

```
# In[69]:
points_cpu = points_gpu.to(device='cpu')
```

Можно также для получения того же результата воспользоваться сокращенными методами `cpu` и `cuda` вместо метода `to`:

```
# In[70]:
points_gpu = points.cuda()  ← По умолчанию переносит на GPU с индексом 0
points_gpu = points.cuda(0)
points_cpu = points_gpu.cpu()
```

Стоит также упомянуть, что с помощью метода `to` можно менять тип данных и их место размещения одновременно, указав в качестве аргументов `device` и `dtype`.

3.10. СОВМЕСТИМОСТЬ С NUMPY

Мы то и дело упоминаем NumPy. И хотя мы не считаем знание этой библиотеки необходимым для чтения данной книги, но настоятельно рекомендуем познакомиться с ней как с повсеместно распространенной в экосистеме Python для целей науки о данных. Тензоры PyTorch можно очень эффективно преобразовывать в массивы NumPy и наоборот. Благодаря этому можно воспользоваться огромными объемами функциональности экосистемы Python, основанной на типах массивов NumPy. Подобная совместимость с массивами NumPy, не требующая копирования кода, возможна благодаря работе системы хранения с буферным протоколом Python (<https://docs.python.org/3/c-api/buffer.html>).

Чтобы получить массив NumPy из нашего тензора `points`, достаточно вызвать:

```
# In[55]:
points = torch.ones(3, 4)
points_np = points.numpy()
points_np

# Out[55]:
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)
```

Этот код возвращает многомерный массив NumPy нужного размера, формы и числового типа. Интересно то, что у возвращаемого массива один и тот же буфер с тензорным хранилищем. Это значит, что выполнение метода `numpy` практически не подразумевает накладных расходов, если данные располагаются в оперативной памяти CPU, а также что модификация массива NumPy ведет к изменениям исходного тензора. Если память под тензор выделяется в GPU, PyTorch копирует содержимое тензора в массив NumPy, расположенный в CPU.

И наоборот, вот так можно получить тензор PyTorch из массива NumPy:

```
# In[56]:
points = torch.from_numpy(points_np)
```

с сохранением той же стратегии совместного использования буфера, которую мы только что описали.

ПРИМЕЧАНИЕ

Числовой тип данных по умолчанию в PyTorch — 32-битное число с плавающей запятой, а в NumPy — 64-битное. Как обсуждалось в подразделе 3.5.2, обычно лучше использовать 32-битные числа с плавающей запятой, так что следует убедиться, что `dtype` наших тензоров после преобразования — `torch.float`.

3.11. ОБОБЩЕННЫЕ ТЕНЗОРЫ ТОЖЕ ТЕНЗОРЫ

Для целей этой книги, да и абсолютного большинства приложений вообще, тензоры — многомерные массивы, точь-в-точь как мы видели выше в этой главе. Если же заглянуть «за кулисы» PyTorch, окажется, что способ хранения данных отличается от API тензоров, обсуждавшегося в разделе 3.6. Любая соответствующая условиям этого API реализация будет считаться тензором!

PyTorch вызывает правильные вычислительные функции вне зависимости от того, размещен ли тензор в памяти CPU или GPU. Достигается это с помощью механизма *диспетчеризации* (*dispatching*), способного обслуживать другие типы тензоров посредством подключения API пользователя к нужным функциям

прикладной части. Разумеется, существуют другие виды тензоров: некоторые специально предназначены для конкретных классов аппаратных устройств (например, TPU от Google), а стратегии представления данных у других отличаются от плотных массивов, с которыми мы сталкивались до сих пор. Например, в разреженных тензорах хранятся только ненулевые значения, а также информация об индексах. Диспетчер PyTorch слева на рис. 3.8 обладает широкими возможностями расширения: дальнейшая коммутация, выполняемая для размещения различных числовых типов данных — на рис. 3.8 (справа), является неизменным правилом реализации, внесенным в настройки любого приложения.

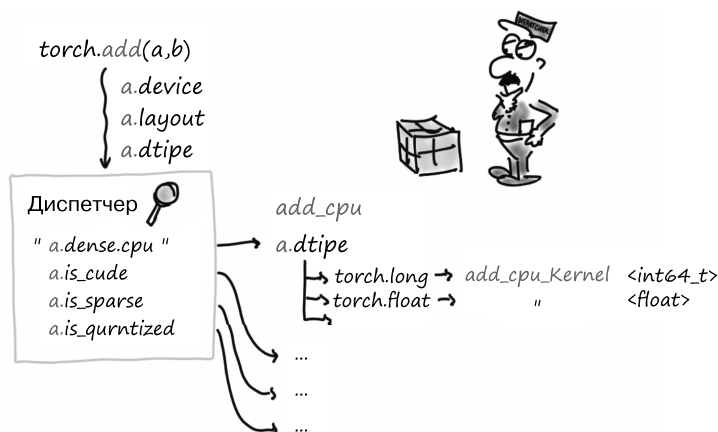


Рис. 3.8. Диспетчер в PyTorch — один из ключевых компонентов инфраструктуры

В главе 15 мы познакомимся с квантованными тензорами, реализованными в виде другого типа тензоров со специализированной вычислительной прикладной частью. Иногда обычные тензоры называют *плотными* (*dense*) или *шаговыми* (*strided*), в отличие от тензоров, для которых используются другие схемы размещения в памяти.

Как это бывает со многими другими вещами, количество видов тензоров в PyTorch растет по мере расширения поддержки PyTorch аппаратного обеспечения и приложений. Можно ожидать возникновения новых видов по мере появления новых способов выражения и выполнения вычислений с помощью PyTorch.

3.12. СЕРИАЛИЗАЦИЯ ТЕНЗОРОВ

Создавать тензоры по ходу дела удобно, но если внутри него ценные данные, желательно сохранить его в файл и загрузить потом обратно. В конце концов, мы же не хотим обучать модель с нуля каждый раз, когда запускаем программу! Для сериализации объектов-тензоров PyTorch использует «за кулисами» `pickle`,

а также специализированный код сериализации для хранилища. Вот как можно сохранить наш тензор `points` в файл `ourpoints.t`:

```
# In[57]:
torch.save(points, '../data/p1ch3/ourpoints.t')
```

Либо можно передать дескриптор файла вместо его названия:

```
# In[58]:
with open('../data/p1ch3/ourpoints.t', 'wb') as f:
    torch.save(points, f)
```

Загрузка тензора `points` обратно тоже выполняется одной строкой кода:

```
# In[59]:
points = torch.load('../data/p1ch3/ourpoints.t')
```

что эквивалентно:

```
# In[60]:
with open('../data/p1ch3/ourpoints.t', 'rb') as f:
    points = torch.load(f)
```

И хотя подобным образом можно быстро сохранять тензоры, если нужно загружать их только в PyTorch, сам по себе формат файла не отличается совместимостью: прочитать тензор с помощью какого-либо еще ПО, помимо PyTorch, не получится. В зависимости от сценария использования, это может и не ограничивать наши возможности, но в противном случае имеет смысл выяснить, как сохранять тензоры совместимым образом. Мы обсудим это в следующем разделе.

3.12.1. Сериализация в HDF5 с помощью h5py

Каждый сценарий использования уникален, но мы подозреваем, что требование совместимости формата хранения тензоров чаще возникает при вводе PyTorch в существующие системы, уже использующие различные библиотеки. Для новых проектов это бывает необходимо реже.

Когда же это необходимо, можно воспользоваться форматом HDF5 и соответствующей библиотекой (<https://www.hdfgroup.org/solutions/hdf5>). HDF5 — переносимый, широко поддерживаемый формат представления сериализованных многомерных массивов, организованный в виде вложенного ассоциативного массива типа «ключ — значение». Python поддерживает формат HDF5 благодаря библиотеке `h5py` (www.h5py.org), принимающей и возвращающей данные в виде массивов NumPy.

Установить библиотеку `h5py` можно с помощью команды:

```
$ conda install h5py
```

Теперь мы можем сохранить тензор `points`, преобразовав его в массив NumPy (без каких-либо накладных расходов, как мы отмечали ранее) и передав функции `create_dataset`:

```
# In[61]:
import h5py

f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'w')
dset = f.create_dataset('coords', data=points.numpy())
f.close()
```

Здесь `'coords'` — ключ для файла в формате HDF5. Могут быть и другие ключи, даже вложенные. В HDF5 интересна возможность индексации набора данных на диске и обращения только к нужным нам элементам. Пусть нам нужно загрузить лишь две последние точки в наборе данных:

```
# In[62]:
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'r')
dset = f['coords']
last_points = dset[-2:]
```

Данные не загружаются до тех пор, пока файл не будет открыт и не потребуется набор данных. Они остаются на диске, пока мы не запрашиваем вторую и последнюю строки набора данных. В этот момент библиотека `h5py` обращается к этим двум столбцам и возвращает объект, напоминающий массив NumPy как поведением, так и API, инкапсулирующий соответствующую часть нашего набора данных.

Благодаря этому можно передать полученный объект функции `torch.from_numpy` и получить непосредственно тензор. Обратите внимание, что в данном случае данные копируются в хранилище тензора:

```
# In[63]:
last_points = torch.from_numpy(dset[-2:])
f.close()
```

В конце загрузки данных мы закрываем файл. Закрытие HDF5-файла делает наборы данных недействительными, так что попытка обратиться к `dset` после этого приведет к ошибке. Если придерживаться приведенной последовательности действий, можно спокойно работать с тензором `last_points`.

3.13. ИТОГИ ГЛАВЫ

Мы охватили все, что необходимо, чтобы представлять все нужные данные в виде чисел с плавающей запятой. Мы обсудили и другие аспекты тензоров: например, создание представлений тензоров, индексацию тензоров другими тензорами и транслирование, упрощающее поэлементные операции над тензорами различных размеров или форм — в случае если это потребуется по ходу работы.

В главе 4 мы узнаем, как представлять настоящие данные в PyTorch. Мы начнем с простых табличных данных и постепенно перейдем к более сложным. А в процессе мы узнаем больше о тензорах.

3.14. УПРАЖНЕНИЯ

1. Создайте тензор `a` из `list(range(9))`. Попробуйте догадаться, какими будут его размер, сдвиг и шаг, а потом проверьте свои догадки.
 - А. Создайте новый тензор с помощью команды `b = a.view(3, 3)`. Что делает функция `view`? Убедитесь, что у тензоров `a` и `b` одно хранилище.
 - Б. Создайте тензор `c = b[1:, 1:]`. Попробуйте догадаться, какими будут его размер, сдвиг и шаг, а потом проверьте свои догадки.
2. Выберите какую-либо математическую операцию, например косинус или корень квадратный. Найдете соответствующую функцию в библиотеке `torch`?
 - А. Примените эту функцию поэлементно к тензору `a`. Почему при этом возвращается ошибка?
 - Б. Какая операция необходима, чтобы функция заработала?
 - В. Существует ли версия нашей функции, которая бы работала с заменой на месте?

3.15. РЕЗЮМЕ

- Нейронные сети преобразуют представления с плавающей запятой в другие представления с плавающей запятой. Начальное и конечное представления обычно понятны для человека, а промежуточные — в меньшей степени.
- Эти представления с плавающей запятой хранятся в тензорах.
- Тензоры представляют собой многомерные массивы и являются основной структурой данных в PyTorch.
- PyTorch — всеобъемлющая библиотека для создания тензоров и выполнения над ними различных операций, в том числе математических.
- Тензоры можно сериализовать на диск и загружать обратно.
- Все операции над тензорами в PyTorch можно выполнять как на CPU, так и на GPU без изменений кода.
- На то, что функции PyTorch производят операции над тензором с заменой на месте, указывает завершающий название знак подчеркивания (например, `Tensor.sqrt_`).

Представление реальных данных с помощью тензоров

В этой главе

- ✓ Представление реальных данных в виде тензоров PyTorch.
- ✓ Работа с различными типами данных.
- ✓ Загрузка данных из файла.
- ✓ Преобразование данных в тензоры.
- ✓ Изменение формы тензоров для их использования в качестве входных сигналов нейросетевых моделей.

Из предыдущей главы мы узнали, что тензоры являются базовыми «кирпичиками» данных в PyTorch. Тензоры играют роль входных и выходных данных нейронных сетей. На самом деле все операции внутри нейронных сетей и во время оптимизации — это операции с тензорами, а все параметры (например, весовые коэффициенты и смещения) нейронной сети — это тензоры. Понимание принципов эффективного выполнения операций над тензорами и доступа по индексам к ним очень важно для успешного применения утилит, подобных PyTorch. Вы уже разобрались с основами тензоров, и ваши навыки работы с ними будут только расти по мере чтения данной книги.

Вопрос, на котором мы уже можем сосредоточиться, звучит так: как представить элемент данных, видеоданных или строку текста в виде тензора, подходящего для модели глубокого обучения? Ответ на него вы и узнаете из этой главы. Мы

рассмотрим различные типы данных, акцентируя внимание на тех, которые понадобятся нам в этой книге, и покажем, как представить их в виде тензоров. Далее мы научимся загружать данные из наиболее распространенных форматов хранения на диске и ознакомимся с их структурой, чтобы разобраться, как подготовить их для обучения нейронной сети. Зачастую исходные данные не совсем подходят для решаемой задачи, так что у нас будет возможность попрактиковаться в выполнении различных операций над тензорами, в том числе нескольких весьма интересных.

Каждый раздел главы будет посвящен какому-либо типу данных, и к каждому из них будет прилагаться собственный набор данных. И хотя мы организовали эту главу исходя из принципа постепенного развития типов данных, вы можете спокойно пропускать отдельные разделы, если вам так хочется.

В оставшейся части книги используется немало объемных пространственных данных и данных изображений в силу их распространенности и удобства отображения в формате книги. Мы также охватим табличные данные, временные ряды и текст, которые могут быть интересны многим нашим читателям. Поскольку одна иллюстрация стоит тысячи слов, мы начнем с данных изображений, а затем продемонстрируем работу с трехмерным массивом медицинских данных, представляющих тело пациента в объемном виде. Далее мы будем работать с табличными данными о винах, которые будут собраны в электронную таблицу. После этого мы обратимся к *упорядоченным* табличным данным на примере набора данных временных рядов из программы аренды велосипедов. Наконец, мы поэкспериментируем с текстами Джейн Остин. Текстовые данные тоже упорядочены, но при работе с ними возникает дополнительная задача представления слов в виде массивов чисел.

В каждом из разделов мы остановимся именно там, где начинается работа специалиста по глубокому обучению: на подаче данных на вход модели. Мы рекомендуем вам сохранить эти наборы данных; они нам пригодятся, когда мы приступим к обучению нейросетевых моделей в следующей главе.

4.1. РАБОТА С ИЗОБРАЖЕНИЯМИ

Появление сверточных нейронных сетей произвело настоящую революцию в сфере машинного зрения (см. <http://mng.bz/zjMa>) и придало совершенно новые возможности основанным на обработке изображений системам. Задачи, требовавшие ранее сложных конвейеров точно подобранных алгоритмических блоков, теперь решаются с неслыханным быстродействием через обучение комплексных сетей на парных примерах данных «входной сигнал/желаемый выходной сигнал». Чтобы поучаствовать в этой революции, необходимо уметь загружать изображения, хранящиеся в наиболее распространенных форматах, и преобразовывать их данные в тензорное представление, в котором различные части изображения организованы подходящим для PyTorch образом.

Изображение при этом представляется в виде набора скалярных значений, расположенных на равномерной сетке с высотой и шириной (в пикселях), например, по одному скалярному значению на каждую точку сетки (пиксель) для изображения в оттенках серого или несколько скалярных значений на каждую точку сетки для представления различных цветов, как мы видели в предыдущих главах, или различных *признаков*, наподобие признака глубины в камере с восприятием глубины сцены.

Отражающие значения для различных пикселей скаляры обычно кодируются 8-битными целыми числами, как в бытовых фотоаппаратах. В медицинских, научных и промышленных приложениях нередко встречается более высокая точность, например 12- или 16-битная, для расширения диапазона или повышения чувствительности в случаях, когда пиксель отражает информацию о физическом свойстве, например о плотности костной ткани, температуре или глубине.

4.1.1. Добавление цветowych каналов

Мы уже упоминали цвета. Существует несколько способов кодирования цветов числами¹. Наиболее распространенный — RGB, в котором цвета определяются тремя числами, отражающими яркость красного, зеленого и синего [цветов]. Цветовой канал можно считать своего рода картой яркости в оттенках серого соответствующего цвета, как если бы мы смотрели на обстановку через чисто красные² очки. На рис. 4.1 показана радуга, в которой каждый из каналов RGB захватывает определенную часть спектра (это, конечно, упрощенный рисунок: в нем опущено, например, представление оранжевой и желтой полос в виде сочетания красной и зеленой).

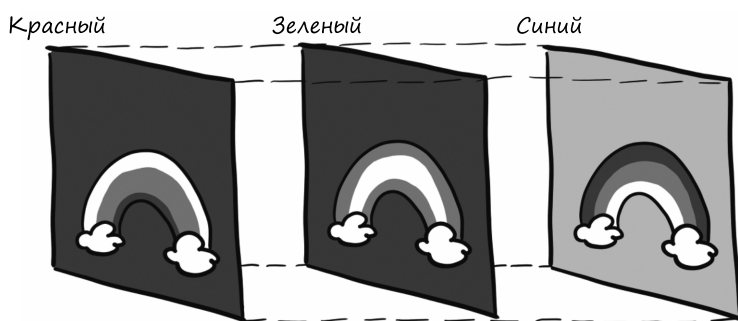


Рис. 4.1. Радуга, разбитая на красный, зеленый и синий каналы

¹ Это еще слабо сказано: https://en.wikipedia.org/wiki/Color_model (версия на русском: https://ru.wikipedia.org/wiki/Цветовая_модель. — *Примеч. пер.*).

² Или зеленые, или синие. — *Примеч. пер.*

Красная полоса радуги — наиболее яркая в красном канале изображения, а синий канал включает на высокой яркости как синюю полосу радуги, так и небо. Обратите внимание также на высокую яркость белых облаков во всех трех каналах.

4.1.2. Загрузка файла изображения

Изображения хранятся во многих файловых форматах, но, к счастью, в Python существует множество способов загрузки изображений. Начнем с загрузки изображения в формате PNG с помощью модуля `imageio` (`code/p1ch4/1_image_dog.ipynb`) (листинг 4.1).

Листинг 4.1. `code/p1ch4/1_image_dog.ipynb`

```
# In[2]:
import imageio

img_arr = imageio.imread('../data/p1ch4/image-dog/bobby.jpg')
img_arr.shape

# Out[2]:
(720, 1280, 3)
```

ПРИМЕЧАНИЕ

В этой главе мы повсюду используем модуль `imageio` благодаря единообразию его API для различных типов данных. Во многих случаях имеет смысл по умолчанию воспользоваться `TorchVision` для работы с данными изображений и видеоданными. Здесь же мы для большей простоты изложения остановимся на `imageio`.

Изображение здесь представляет собой подобный массиву NumPy объект с тремя измерениями: два пространственных (ширина и высота) и третье измерение, соответствующее красному, зеленому и синему каналам. Для получения тензора PyTorch подойдет любая библиотека, выдающая массивы NumPy. Следует только учесть расстановку измерений. Модули PyTorch, работающие с изображениями, требуют от тензоров схемы измерений $C \times H \times W$ (каналы, высота и ширина соответственно).

4.1.3. Изменение схемы расположения

Для получения нужной нам схемы расположения можно воспользоваться методом `permute` тензора, указав в качестве параметров старые измерения для каждого из новых. При входном тензоре вида $H \times W \times C$, полученном нами ранее, нужную схему расположения можно получить, указав сначала измерение 2, а затем 0 и 1:

```
# In[3]:
img = torch.from_numpy(img_arr)
out = img.permute(2, 0, 1)
```

Мы уже видели это ранее, но обратите внимание, что эта операция не копирует данные тензора, вместо этого `out` использует то же самое хранилище, что и `img`, только меняя информацию о размере и шаге на уровне тензора. Удобство такой операции в низких затратах ресурсов, но сразу предостерегаем вас: изменение пикселя в `img` приведет к изменениям в `out`.

Обратите также внимание, что в других фреймворках глубокого обучения применяются другие схемы расположения. Например, изначально в TensorFlow измерение каналов было последним, в результате чего получалась схема $H \times W \times C$ (сейчас он поддерживает несколько схем). С точки зрения низкоуровневой производительности у такой стратегии есть свои достоинства и недостатки, но для наших целей это неважно, достаточно менять форму тензоров нужным образом.

До сих пор мы работали с отдельным изображением. Следуя стратегии, применявшейся нами для предыдущих типов данных, чтобы создать входной набор данных для наших нейронных сетей из нескольких изображений, мы сохраняем изображения в батчах по первому измерению и получаем тензор вида $N \times C \times H \times W$.

Несколько более эффективная альтернатива использованию для создания тензора `stack` — выделить заранее память под тензор нужного размера, а затем заполнить его загруженными из каталога изображениями следующим образом:

```
# In[4]:
batch_size = 3
batch = torch.zeros(batch_size, 3, 256, 256, dtype=torch.uint8)
```

Как видим, наш батч будет состоять из трех RGB-изображений по 256 пикселей высотой и 256 пикселей шириной. Обратите внимание на тип этого тензора: мы предполагаем, что каждый из цветов будет представлен 8-битным целым числом, как в большинстве форматов фотографий из стандартных бытовых фотоаппаратов. Теперь можно загрузить все изображения в формате PNG из входного каталога и сохранить их в тензоре:

```
# In[5]:
import os
```

```
data_dir = '../data/p1ch4/image-cats/'
filenames = [name for name in os.listdir(data_dir)
               if os.path.splitext(name)[-1] == '.png']
for i, filename in enumerate(filenames):
    img_arr = imageio.imread(os.path.join(data_dir, filename))
    img_t = torch.from_numpy(img_arr)
    img_t = img_t.permute(2, 0, 1)
    img_t = img_t[:3]
    batch[i] = img_t
```

Мы сохраняем только первые три канала. Иногда в изображениях встречается также альфа-канал, отвечающий за прозрачность, но нашей нейронной сети достаточно входных данных вида RGB

4.1.4. Нормализация данных

Мы уже упоминали ранее, что роль входных сигналов нейронных сетей обычно играют тензоры с плавающей запятой. Нейронные сети демонстрируют наилучшее качество обучения, когда входные данные находятся в диапазоне примерно от 0 до 1 или от -1 до 1 (из-за того, как описаны их структурные элементы).

Так что обычно требуется привести тип тензора к числам с плавающей запятой и нормализовать значения пикселей. Приведение к числам с плавающей запятой будет простой задачей, в то время как нормализация доставит много проблем, поскольку зависит от выбираемого нами для приведения к интервалу от 0 до 1 (или от -1 до 1) диапазона входных данных. Один из вариантов: просто разделить значения пикселей на 255 (максимальное число, которое можно представить с помощью беззнакового 8-битного числа):

```
# In[6]:
batch = batch.float()
batch /= 255.0
```

Еще одна возможность: вычислить среднее значение и стандартное отклонение входных данных и масштабировать их так, чтобы у выходного сигнала было нулевое среднее значение и единичное стандартное отклонение по каждому каналу:

```
# In[7]:
n_channels = batch.shape[1]
for c in range(n_channels):
    mean = torch.mean(batch[:, c])
    std = torch.std(batch[:, c])
    batch[:, c] = (batch[:, c] - mean) / std
```

ПРИМЕЧАНИЕ

Здесь мы нормализуем только один батч изображений, поскольку не умеем пока что работать со всем набором данных сразу. При работе с изображениями рекомендуется заранее вычислять среднее значение и стандартное отклонение по всем обучающим данным, а затем вычитать первое и делить на второе фиксированное, заранее вычисленное значение. Мы уже встречали эту методику при предварительной обработке для классификатора изображений в подразделе 2.1.4.

Над входными данными можно производить еще несколько операций, в частности геометрические преобразования (повороты, масштабирование и обрезку). Они могут пригодиться для обучения или потребоваться для подгонки входных данных к требованиям сети, например, относительно размера изображения. Мы столкнемся со многими из этих стратегий в разделе 12.6. А пока просто помните о доступных возможностях обработки изображений.

4.2. ТРЕХМЕРНЫЕ ИЗОБРАЖЕНИЯ: ОБЪЕМНЫЕ ПРОСТРАНСТВЕННЫЕ ДАННЫЕ

Мы научились загружать и отображать двумерные изображения, например те, которые мы делаем с помощью фотоаппарата. В некоторых контекстах, например в приложениях для медицинской визуализации, в частности КТ (компьютерной томографии), обычно приходится иметь дело с последовательностями изображений, сложенных вдоль оси «голова — ноги», каждое из которых соответствует срезу человеческого тела. При компьютерной томографии яркость отражает плотность различных частей человеческого тела: легких, жира, жидкости, мышц и костной ткани — в порядке роста плотности, показываемой от темного к светлому при отображении на рабочей станции врача. Плотность в каждой точке вычисляется на основе интенсивности рентгеновского излучения, достигающего датчика после прохождения через тело, с помощью достаточно сложных математических расчетов для расшифровки исходных данных с датчика в объемные данные.

КТ-снимки содержат только один канал яркости, подобно изображениям в оттенках серого. Это значит, что зачастую измерение каналов не учитывается в исходных форматах данных, и поэтому аналогично предыдущему разделу исходные данные обычно содержат только три измерения. Складывая отдельные двумерные срезы в трехмерный тензор, можно сформировать объемные пространственные данные, отражающие трехмерную анатомию человека. В отличие от увиденного нами на рис. 4.1, дополнительное измерение на рис. 4.2 отражает сдвиг в физическом пространстве, а не конкретную полосу видимого спектра.

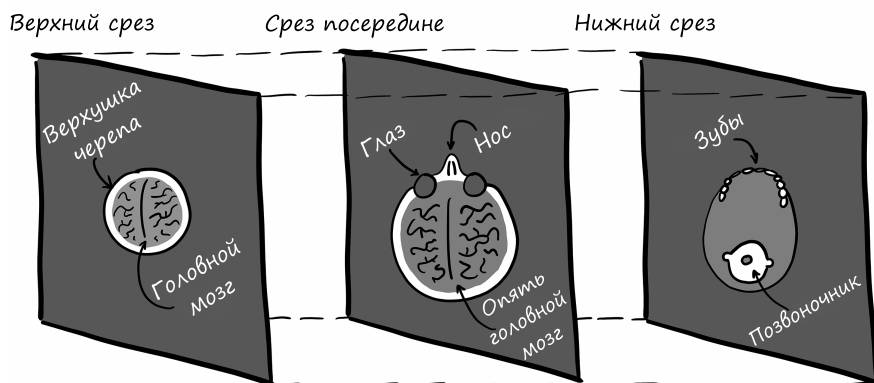


Рис. 4.2. Срезы КТ-снимка от макушки до подбородка

Вторая часть этой книги посвящена задаче выполнения медицинских снимков на практике, так что мы не станем сейчас вникать в нюансы форматов данных

медицинских снимков. Пока достаточно упомянуть, что никаких принципиальных различий между тензорами, содержащими объемные пространственные данные и данные изображений, нет. Просто появляется дополнительное измерение, *глубина* (*depth*), вслед за измерением *каналов*, и получается пятимерный тензор формы $N \times C \times D \times H \times W$.

4.2.1. Загрузка данных в специализированном формате

Загрузим пример КТ-снимка с помощью функции `volread` из модуля `imageio`, принимающей в качестве аргумента каталог и собирающей все файлы в формате DICOM (Digital Imaging and Communications in Medicine, «цифровые изображения и обмен данными в медицине»)¹ в трехмерный массив NumPy (`code/p1ch4/2_volumetric_ct.ipynb`) (листинг 4.2).

Листинг 4.2. `code/p1ch4/2_volumetric_ct.ipynb`

```
# In[2]:
import imageio

dir_path = "../data/p1ch4/volumetric-dicom/2-LUNG 3.0B70f-04083"
vol_arr = imageio.volread(dir_path, 'DICOM')
vol_arr.shape

# Out[2]:
Reading DICOM (examining files): 1/99 files (1.0%99/99 files (100.0%)
Found 1 correct series.
Reading DICOM (loading data): 31/99 (31.392/99 (92.999/99 (100.0%)

(99, 512, 512)
```

Как и в подразделе 4.1.3, схема расположения отличается от ожидаемого PyTorch из-за отсутствия информации о канале. Так что нужно дать место для измерения `channel` с помощью `unsqueeze`:

```
# In[3]:
vol = torch.from_numpy(vol_arr).float()
vol = torch.unsqueeze(vol, 0)

vol.shape

# Out[3]:
torch.Size([1, 99, 512, 512])
```

Теперь можно сформировать пятимерный набор данных, разместив один за другим несколько срезов по измерению `batch`, подобно тому как мы делали в предыдущем разделе. В части II нас ждет еще много данных КТ-снимков.

¹ Из набора CPTAC-LSCC Cancer Imaging Archive: <http://mng.bz/K21K>.

4.3. ПРЕДСТАВЛЕНИЕ ТАБЛИЧНЫХ ДАННЫХ

Самая простая форма данных, с которой мы сталкиваемся при работе с машинным обучением, поступает из электронных таблиц, CSV-файлов и баз данных. Вне зависимости от способа хранения это все равно будет таблицей, каждая строка которой соответствует примеру данных (или записи), а каждый столбец содержит какой-либо элемент информации о нашем примере данных.

Сначала мы будем считать, что порядок примеров данных в таблице неважен: таблица представляет собой набор независимых примеров, в отличие, к примеру, от временных рядов, в которых примеры данных связаны временным измерением.

Столбцы могут содержать числовые значения (например, температуру в конкретных местах) или метки (например, строковые значения, отражающие какой-либо атрибут примера данных, например «синий»). Следовательно, табличные данные обычно неоднородны: типы различных столбцов различаются. Один столбец может содержать вес яблок, а другой — кодировку их цвета в виде меток.

Тензоры PyTorch, с другой стороны, однородны. Информация в PyTorch обычно кодируется числами, чаще всего с плавающей запятой (хотя поддерживаются целочисленные и булевы типы данных). Такое числовое кодирование было выбрано осознанно, поскольку нейронные сети — это математические объекты, принимающие действительные числа на входе и генерирующие действительные числа на выходе путем последовательного применения умножений матриц и нелинейных функций.

4.3.1. Реальный набор данных

Наше первое задание как специалистов по глубокому обучению — закодировать реальные неоднородные данные в тензор, содержащий числа с плавающей запятой, подходящий для потребления нейронной сетью. В интернете можно найти много свободно доступных табличных наборов данных, см., например, <https://github.com/caesar0301/awesome-public-datasets>. Начнем с чего-нибудь интересного, например с вина! Набор данных по вкусовым качествам вина представляет собой общедоступную таблицу, содержащую химические характеристики примеров данных *винью-верде*, вина с севера Португалии, а также органолептическую оценку качества. Набор данных для белых вин можно скачать здесь: <http://mng.bz/900l>. Для удобства мы также создали копию этого набора данных в репозитории Git книги, в каталоге `data/p1ch4/tabular-wine`.

Этот файл содержит разделенный запятыми набор значений, организованных по 12 столбцам, которым предшествует строка заголовка с названиями столбцов. Первые 11 столбцов содержат значения химических величин,

а последний — органолептическую оценку качества, от 0 (очень плохо) до 10 (великолепно). Вот названия столбцов в порядке их появления в наборе данных:

fixed acidity
volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH
sulphates
alcohol
quality

Одна из возможных задач машинного обучения с этим набором данных — предсказание оценки вкусовых качеств вина по одним химическим характеристикам. Впрочем, не беспокойтесь, машинное обучение не заменит дегустацию вина в ближайшем будущем. Откуда-то же нужно получать обучающие данные! Как можно видеть на рис. 4.3, мы хотим найти связь между одним из столбцов химических характеристик в наших данных и столбцом вкусовых качеств. В данном случае можно ожидать роста вкусовых качеств при снижении содержания серы.

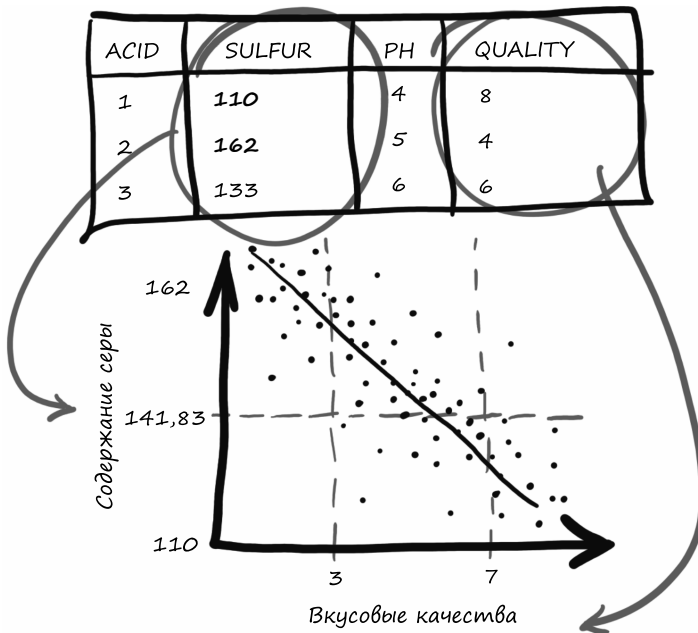


Рис. 4.3. (Предполагаемая) связь между содержанием серы и вкусовыми качествами вина

4.3.2. Загрузка тензора данных по вину

Прежде чем мы сможем это сделать, впрочем, необходимо не просто открыть файл в текстовом редакторе, а просмотреть данные более удобным способом. Давайте взглянем, как можно загрузить данные с помощью языка Python и преобразовать их в тензор PyTorch. Python предоставляет несколько вариантов быстрой загрузки CSV-файла. Три наиболее распространенных:

- поставляемый в составе установочного пакета Python модуль `csv`;
- NumPy;
- Pandas.

Наиболее рациональный по занимаемому времени и объему используемой памяти — третий вариант. Однако мы не хотели бы заставлять вас изучать дополнительную библиотеку просто потому, что нам нужно загрузить файл. А поскольку мы уже познакомили вас с библиотекой NumPy в предыдущем разделе, а PyTorch прекрасно умеет взаимодействовать с NumPy, мы воспользуемся ею. Давайте загрузим наш файл и преобразуем полученный массив NumPy в тензор PyTorch (`code/p1ch4/3_tabular_wine.ipynb`) (листинг 4.3).

Листинг 4.3. `code/p1ch4/3_tabular_wine.ipynb`

```
# In[2]:
import csv
wine_path = "../data/p1ch4/tabular-wine/winequality-white.csv"
wineq_numpy = np.loadtxt(wine_path, dtype=np.float32, delimiter=";",
                        skiprows=1)

wineq_numpy

# Out[2]:
array([[ 7. , 0.27, 0.36, ..., 0.45, 8.8 , 6. ],
       [ 6.3 , 0.3 , 0.34, ..., 0.49, 9.5 , 6. ],
       [ 8.1 , 0.28, 0.4 , ..., 0.44, 10.1 , 6. ],
       ...,
       [ 6.5 , 0.24, 0.19, ..., 0.46, 9.4 , 6. ],
       [ 5.5 , 0.29, 0.3 , ..., 0.38, 12.8 , 7. ],
       [ 6. , 0.21, 0.38, ..., 0.32, 11.8 , 6. ]], dtype=float32)
```

Здесь мы просто указываем требуемый тип двумерного массива (32-битные числа с плавающей запятой), разделитель значений в каждой из строк и то, что первую строку с названиями столбцов читать не нужно. Давайте проверим, что были прочитаны все данные.

```
# In[3]:
col_list = next(csv.reader(open(wine_path), delimiter=';'))

wineq_numpy.shape, col_list

# Out[3]:
```



```
((4898, 12),
 ['fixed acidity',
 'volatile acidity',
 'citric acid',
 'residual sugar',
 'chlorides',
 'free sulfur dioxide',
 'total sulfur dioxide',
 'density',
 'pH',
 'sulphates',
 'alcohol',
 'quality'])
```

и займемся преобразованием массива NumPy в тензор PyTorch:

```
# In[4]:
wineq = torch.from_numpy(wineq_numpy)
wineq.shape, wineq.dtype

# Out[4]:
(torch.Size([4898, 12]), torch.float32)
```

Теперь у нас есть объект `torch.Tensor` с плавающей запятой, содержащий все столбцы данных, включая последний, с оценками вкусовых качеств.

НЕПРЕРЫВНЫЕ, ПОРЯДКОВЫЕ И КАТЕГОРИАЛЬНЫЕ ЗНАЧЕНИЯ

При попытке разобраться в данных следует учитывать существование трех различных видов числовых величин¹. Первая разновидность — *непрерывные* (*continuous*) величины. Их удобнее всего представлять в виде числовых значений. Они строго упорядочены, и смысл разности двух величин строго определен. Смысл утверждения «посылка А на 2 кг тяжелее, чем посылка В» или «посылка В прошла расстояние на 100 км большее, чем А» четко определен вне зависимости от того, весит посылка А 3 кг или 10, или того, прошла посылка В 200 км или 2000. Величины, подсчитываемые или измеряемые с помощью каких-либо единиц измерения, обычно непрерывные. В литературе обычно проводится более подробная классификация непрерывных величин: в предыдущих примерах имели смысл фразы вида «А вдвое тяжелее В» или «А втрое дальше В». Поэтому о таких величинах говорится как о находящихся на *шкале отношений* (*ratio scale*). У времени суток, с другой стороны, тоже есть понятие разности, но нельзя сказать, что 6:00 вдвое позднее, чем 3:00, так что для времени суток существует только *интервальная шкала* (*interval scale*).

¹ В качестве отправной точки для более подробного обсуждения загляните по адресу https://en.wikipedia.org/wiki/Level_of_measurement (русскаяязычная версия: <https://ru.wikipedia.org/wiki/Шкала>).

Далее существуют *порядковые (ordinal)* величины. Они тоже отличаются строгой упорядоченностью, как и непрерывные, но фиксированное отношение между величинами отсутствует. Хороший пример: упорядоченность маленького, среднего и большого стакана напитка, где маленькому соответствует значение 1, среднему — 2, а большому — 3. Большой стакан больше среднего, так же как 3 больше 2, но мы не знаем, *насколько* именно больше. Если перевести наши 1, 2, 3 в фактические значения объема (скажем, 200, 300 и 600 мл), они станут интервальными значениями. Важно помнить, что нельзя «производить арифметические действия» над такими значениями, кроме их упорядочения; при усреднении большого = 3 и маленького = 1 стаканов не получится средний!

Наконец, у значений *категориальных (categorical)* величин отсутствует как упорядоченность, так и числовой смысл. Они представляют собой просто перечисления возможностей, которым присвоены произвольные числа. Хороший пример: вода = 1, кофе = 2, содовая = 3 и молоко = 4. Никакой особой логики в том, что вода предшествует молоку, нет; у них просто должны быть отдельные значения, чтобы их можно было различать. Можно присвоить кофе значение 10, а молоку –3, и ничего особо не поменяется (хотя для унитарного кодирования и вложений слов, которые мы обсудим в подразделе 4.5.4, лучше присваивать значения в диапазоне 0... N – 1). А поскольку эти числовые значения не несут смысла, в этом случае говорят о *номинальной шкале (nominal scale)*.

4.3.3. Представление оценок

Можно рассматривать оценку как непрерывную величину, хранить ее в виде вещественного числа и решать задачу регрессии или же рассматривать ее как метку и пытаться предсказать метку, исходя из химического анализа в задаче классификации. При любом подходе оценки обычно исключаются из тензора входных данных и хранятся в отдельном тензоре, чтобы использовать оценки в качестве эталонных данных без подачи их на вход нашей модели:

```
# In[5]:
data = wineq[:, :-1]  ← Выбираем все строки и все столбцы, кроме последнего
data, data.shape

# Out[5]:
(tensor([[ 7.00, 0.27, ..., 0.45, 8.80],
         [ 6.30, 0.30, ..., 0.49, 9.50],
         ...,
         [ 5.50, 0.29, ..., 0.38, 12.80],
         [ 6.00, 0.21, ..., 0.32, 11.80]]), torch.Size([4898, 11]))

# In[6]:
target = wineq[:, -1] ← Выбираем все строки и последний столбец
target, target.shape

# Out[6]:
(tensor([6., 6., ..., 7., 6.]), torch.Size([4898]))
```

Если же преобразовывать тензор `target` в тензор меток, возможны два варианта в зависимости от выбранной стратегии и того, для чего предназначаются категориальные данные. Первый заключается в том, что можно просто рассматривать метки как целочисленный вектор оценок:

```
# In[7]:
target = wineq[:, -1].long()
target

# Out[7]:
tensor([6, 6, ..., 7, 6])
```

В случае же, когда `target` содержит строковые метки, например *wine color*, можно задать целочисленное значение для каждого из строковых значений и следовать тому же подходу.

4.3.4. Быстрое кодирование

Еще один подход — *быстрое кодирование (one-hot encoding)* оценок, то есть кодирование каждой из десяти оценок вектором из десяти элементов, в котором все элементы, кроме одного (на различной позиции для каждой из оценок), равны 0. При этом оценке 1 будет соответствовать вектор (1,0,0,0,0,0,0,0,0,0), а оценке 5 — вектор (0,0,0,0,1,0,0,0,0,0) и т. д. Обратите внимание, что совпадение оценки с индексом ненулевого элемента — это чисто случайное явление: можно как угодно перетасовать это отображение, и ничего с точки зрения классификации не поменяется.

Существует заметное различие этих двух подходов. Хранение оценок вкусовых качеств вина в целочисленном векторе влечет их упорядоченность, что в данном случае совершенно неуместно, ведь оценка 1 ниже оценки 4. Это также подразумевает наличие расстояния между оценками: то есть расстояние между 1 и 3 равно расстоянию между 2 и 4. Если для нашей величины это подходит — замечательно. Если же, напротив, оценки совершенно дискретны, например, означают сорт винограда, то быстрое кодирование подойдет намного лучше, поскольку не подразумевает упорядоченности или расстояния. Быстрое кодирование также подходит для количественных оценок, когда дробные значения между двумя целочисленными оценками, например 2,4, бессмысленны в конкретном приложении, то есть когда оценка вида либо *то*, либо *другое*.

Для быстрого кодирования можно воспользоваться методом `scatter_`, заполняющим тензор значениями из исходного тензора по указанным в качестве аргумента индексам:

```
# In[8]:
target_onehot = torch.zeros(target.shape[0], 10)
target_onehot.scatter_(1, target.unsqueeze(1), 1.0)
```

```
# Out[8]:
tensor([[0., 0., ..., 0., 0.],
        [0., 0., ..., 0., 0.],
        ...,
        [0., 0., ..., 0., 0.],
        [0., 0., ..., 0., 0.]])
```

Взглянем, что делает метод `scatter_`. Прежде всего видим, что его название заканчивается символом подчеркивания. Как вы уже знаете из предыдущей главы, по условиям PyTorch это означает, что данный метод не возвращает новый тензор, а изменяет имеющийся. Аргументы метода `scatter_` следующие:

- измерение, по которому указываются следующие два аргумента;
- тензор-столбец с индексами заполняемых элементов;
- тензор с элементами для распределения или одно заполняющее скалярное значение (в данном случае 1).

Другими словами, предыдущий вызов гласит: «Для каждой строки нужно воспользоваться индексом целевой метки (совпадающим с оценкой в нашем случае) в качестве индекса столбца для задания значения 1,0». В результате получается тензор, кодирующий категориальную информацию.

Размерность второго аргумента метода `scatter_`, тензора индексов, должна совпадать с заполняемым тензором. Поскольку размерность тензора `target_onehot` равна 2 (4898×10), необходимо добавить еще одно фиктивное измерение в `target` с помощью функции `unsqueeze`:

```
# In[9]:
target_unsqueezed = target.unsqueeze(1)
target_unsqueezed

# Out[9]:
tensor([[6],
        [6],
        ...,
        [7],
        [6]])
```

В результате вызова `unsqueeze` добавляется *одноэлементное* измерение, превращая одномерный тензор в 4898 элементов в двумерный тензор размером (4898×1), без изменения его содержимого: никаких дополнительных элементов не добавляется, так как мы просто решили использовать еще один индекс для доступа к элементам. То есть к первому элементу `target` можно обращаться `target[0]`, а к первому элементу его аналога, к которому была применена `unsqueeze`, — `target_unsqueezed[0,0]`.

PyTorch дает возможность непосредственно использовать индексы классов в качестве целей при обучении нейронной сети. Однако для использования оценок

в качестве категориального входного сигнала сети необходимо преобразовать его в кодированный в быстром режиме тензор.

4.3.5. Когда считать данные категориальными

Мы уже умеем обрабатывать как непрерывные, так и категориальные данные. Наверное, вы задаете себе вопрос: «А что делать в случае порядковых данных, упомянутых во врезке выше?» Общего рецепта на все случаи жизни не существует; чаще всего с подобными данными обращаются как с категориальными (забывая про упорядоченность и надеясь на то, что модель сможет учесть это во время обучения, если категорий всего несколько) или непрерывными (через ввод в модель произвольно выбранной меры расстояния). Мы рассмотрим второй из этих сценариев на примере ситуации с погодой на рис. 4.5. Подытожим наше отображение данных в небольшой блок-схеме на рис. 4.4.

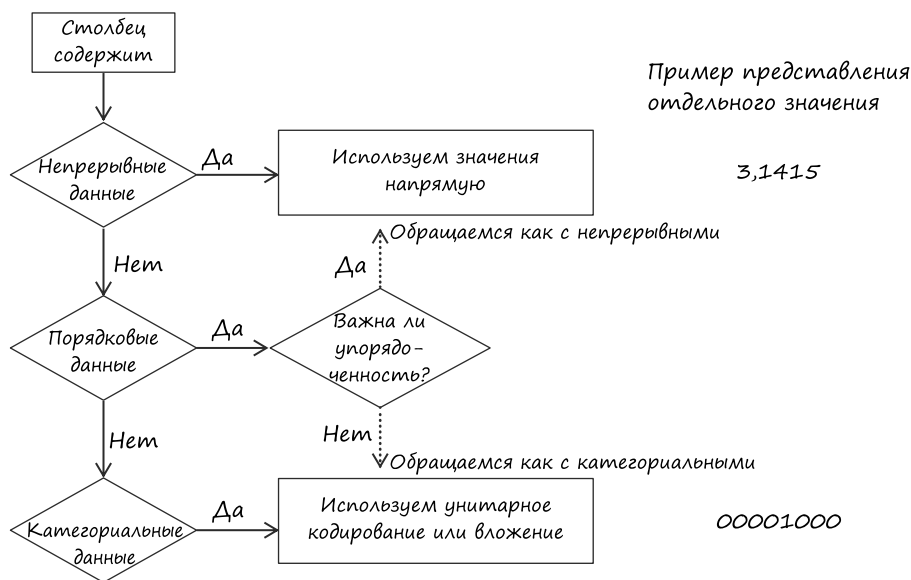


Рис. 4.4. Что делать со столбцами, содержащими непрерывные, порядковые и категориальные данные

Вернемся к нашему тензору `data`, содержащему 11 величин, относящихся к химическому анализу. Для операций над нашими данными в форме тензора мы можем воспользоваться функциями API Tensor PyTorch. Сначала вычислим среднее значение и стандартное отклонение каждого из столбцов:

```
# In[10]:
data_mean = torch.mean(data, dim=0)
```

```
data_mean
# Out[10]:
tensor([[6.85e+00, 2.78e-01, 3.34e-01, 6.39e+00, 4.58e-02, 3.53e+01,
        1.38e+02, 9.94e-01, 3.19e+00, 4.90e-01, 1.05e+01]])

# In[11]:
data_var = torch.var(data, dim=0)
data_var
# Out[11]:
tensor([[7.12e-01, 1.02e-02, 1.46e-02, 2.57e+01, 4.77e-04, 2.89e+02,
        1.81e+03, 8.95e-06, 2.28e-02, 1.30e-02, 1.51e+00]])
```

В данном случае `dim=0` означает выполнение свертки по измерению 0. В этот момент можно нормализовать данные посредством вычитания среднего значения и деления на стандартное отклонение для улучшения процесса обучения (мы обсудим этот вопрос подробнее в подразделе 5.4.4):

```
# In[12]:
data_normalized = (data - data_mean) / torch.sqrt(data_var)
data_normalized
# Out[12]:
tensor([[ 1.72e-01, -8.18e-02, ..., -3.49e-01, -1.39e+00],
        [-6.57e-01, 2.16e-01, ..., 1.35e-03, -8.24e-01],
        ...,
        [-1.61e+00, 1.17e-01, ..., -9.63e-01, 1.86e+00],
        [-1.01e+00, -6.77e-01, ..., -1.49e+00, 1.04e+00]])
```

4.3.6. Поиск пороговых значений

А теперь проанализируем данные и попробуем найти простой способ с первого взгляда различить хорошие и плохие вина. Прежде всего попробуем определить, какие строки в `target` соответствуют оценкам 3 и менее:

```
# In[13]:
bad_indexes = target <= 3
bad_indexes.shape, bad_indexes.dtype, bad_indexes.sum()

# Out[13]:
(torch.Size([4898]), torch.bool, tensor(20))
```

PyTorch также предоставляет функции сравнения, здесь это `torch.le(target, 3)`. Но использование операторов кажется хорошим вариантом

Обратите внимание, что только 20 записей `bad_indexes` равны `True`! С помощью одной из возможностей PyTorch — *расширенного доступа по индексу* (*advanced indexing*) — можно индексировать тензор `data` тензором с типом данных `torch.bool`. Это приведет к фильтрации `data` до одних только элементов (строк), соответствующих `True` в тензоре индексов. Форма `bad_indexes` такая же, как у `target`, а значение `False` или `True` зависит от результата сравнения нашего порогового значения с каждым из элементов исходного тензора `target`:

```
# In[14]:
bad_data = data[bad_indexes]
bad_data.shape
```

```
# Out[14]:
torch.Size([20, 11])
```

Обратите внимание, что новый тензор `bad_data` содержит 20 строк, столько же, сколько строк со значением `True` в тензоре `bad_indexes`. И все 11 столбцов. Теперь мы можем приступить к получению информации о винах, сгруппированной по категориям хороших, посредственных и плохих вин. Давайте вычислим `.mean()` каждого из столбцов:

```
# In[15]:
bad_data = data[target <= 3]
mid_data = data[(target > 3) & (target < 7)]
good_data = data[target >= 7]

bad_mean = torch.mean(bad_data, dim=0)
mid_mean = torch.mean(mid_data, dim=0)
good_mean = torch.mean(good_data, dim=0)
```

Для логических массивов NumPy и тензоров PyTorch оператор & выполняет логическую операцию «И»

```
for i, args in enumerate(zip(col_list, bad_mean, mid_mean, good_mean)):
    print('{:2} {:20} {:.6.2f} {:.6.2f} {:.6.2f}'.format(i, *args))
```

```
# Out[15]:
0 fixed acidity          7.60    6.89    6.73
1 volatile acidity      0.33    0.28    0.27
2 citric acid           0.34    0.34    0.33
3 residual sugar        6.39    6.71    5.26
4 chlorides             0.05    0.05    0.04
5 free sulfur dioxide   53.33   35.42   34.55
6 total sulfur dioxide 170.60 141.83 125.25
7 density               0.99    0.99    0.99
8 pH                   3.19    3.18    3.22
9 sulphates            0.47    0.49    0.50
10 alcohol              10.34   10.26   11.42
```

Похоже, мы наткнулись на что-то интересное: на первый взгляд у плохих вин выше общее содержание сернистого ангидрида, помимо прочих отличий. Мы можем установить пороговое значение по общему содержанию сернистого ангидрида, чтобы отличать хорошие вина от плохих. Давайте получим индексы, по которым столбец с общим содержанием сернистого ангидрида содержит значение меньше среднего, вычисленного ранее, следующим образом:

```
# In[16]:
total_sulfur_threshold = 141.83
total_sulfur_data = data[:,6]
predicted_indexes = torch.lt(total_sulfur_data, total_sulfur_threshold)

predicted_indexes.shape, predicted_indexes.dtype, predicted_indexes.sum()

# Out[16]:
(torch.Size([4898]), torch.bool, tensor(2727))
```

Следовательно, такое пороговое значение подразумевает, что более половины вин окажется высокого качества. Далее нам необходимо получить индексы по-настоящему хороших вин:

```
# In[17]:
actual_indexes = target > 5

actual_indexes.shape, actual_indexes.dtype, actual_indexes.sum()

# Out[17]:
(torch.Size([4898]), torch.bool, tensor(3258))
```

То, что по-настоящему хороших вин примерно на 500 больше, чем предсказывает наше пороговое значение, будет убедительным доказательством его несовершенства. Теперь необходимо проверить, совпадают ли наши предсказания с настоящим рейтингом вин. Мы произведем операцию логического «И» между индексами наших предсказаний и настоящими индексами хороших вин (помните, что все они представляют собой просто массивы нулей и единиц) и воспользуемся этим пересечением множеств вин, чтобы определить, насколько хорошие результаты демонстрирует наша модель:

```
# In[18]:
n_matches = torch.sum(actual_indexes & predicted_indexes).item()
n_predicted = torch.sum(predicted_indexes).item()
n_actual = torch.sum(actual_indexes).item()

n_matches, n_matches / n_predicted, n_matches / n_actual

# Out[18]:
(2018, 0.74000733406674, 0.6193984039287906)
```

Мы правильно оценили около 2000 вин! Поскольку мы предсказали 2700 вин, получается 74%-ная вероятность, что наше предсказание высокого качества вина окажется правильным. К сожалению, хороших вин на самом деле 3200, и мы определили только 61 % из них. Что ж, мы получили, что заслуживали: наша оценка лишь немногим лучше случайной! Конечно, это очень наивная модель: разумеется, на вкусовые качества вина влияет множество величин, а взаимосвязь между значениями этих величин и конечным результатом (который может представлять собой настоящую оценку, а не преобразованную в двоичную форму), вероятно, намного сложнее простого порогового значения одной величины.

Разумеется, даже простая нейронная сеть может преодолеть все эти ограничения, как и многие другие основные методы машинного обучения. После следующих двух глав, когда мы научимся создавать нейронные сети с самого начала, у нас будут все необходимые инструменты для решения этой задачи. В главе 12 мы также вернемся к вопросу о том, как лучше ранжировать наши результаты. А пока что обратим наше внимание на другие типы данных.

4.4. ВРЕМЕННЫЕ РЯДЫ

В предыдущем разделе мы охватили вопрос представления данных, организованных в виде двумерной таблицы. Как мы отмечали, все строки в этой таблице были независимы, их порядок роли не играл. Или, что эквивалентно, ни один столбец не содержал информацию о том, какие строки следуют ранее, а какие — позднее.

Если вернуться к набору данных о винах, то в нем мог бы быть столбец «год», который позволил бы нам изучить улучшение вкусовых качеств вин с годами. К сожалению, у нас таких данных нет, но мы усердно занимаемся ручным сбором примеров данных, бутылка за бутылкой (материалы для второго издания нашей книги). Тем временем, мы переключимся на другой интересный набор данных: данные из системы аренды велосипедов (Вашингтон, округ Колумбия), включающие почасовое количество арендуемых велосипедов в 2011–2012 годах в системе Capital Bikeshare вместе с информацией о погоде и времени года (доступно здесь: <http://mng.bz/jgOx>). Наша цель состоит в том, чтобы преобразовать «плоский», двумерный набор данных в трехмерный, как показано на рис. 4.5.

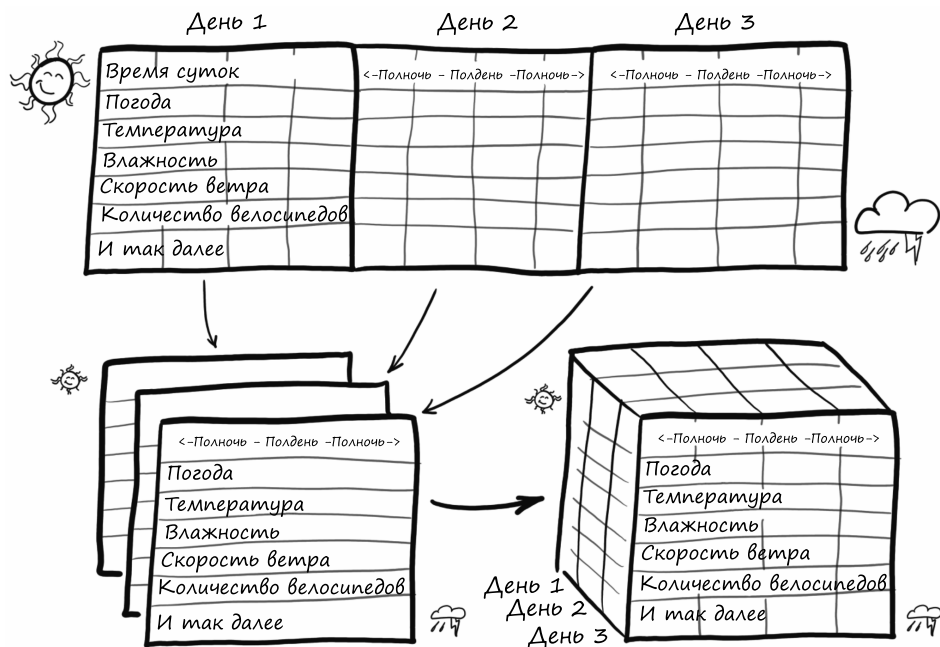


Рис. 4.5. Преобразование одномерного многоканального набора данных в двумерный многоканальный набор данных путем разведения даты и времени всех примеров данных по отдельным осям координат

4.4.1. Добавляем измерение времени

Каждая строка в исходных данных соответствует данным за один отдельный час (на рис. 4.5 приведена транспонированная версия, которую удобнее размещать на печатной странице). Мы хотели бы изменить эту организацию данных «по строке на час», чтобы одна ось координат увеличивалась со скоростью один день на единицу индекса, а другая отражала час суток (независимо от даты). Третья ось координат будет соответствовать различным столбцам данных (погода, температура и т. д.).

Загрузим данные (`code/p1ch4/4_time_series_bikes.ipynb`) (листинг 4.4).

Листинг 4.4. `code/p1ch4/4_time_series_bikes.ipynb`

```
# In[2]:
bikes_numpy = np.loadtxt(
    "../data/p1ch4/bike-sharing-dataset/hour-fixed.csv",
    dtype=np.float32,
    delimiter=",",
    skiprows=1,
    converters={1: lambda x: float(x[8:10])})
bikes = torch.from_numpy(bikes_numpy)
bikes
```

Преобразует строки даты в числа, соответствующие дню месяца в столбце 1

```
# Out[2]:
tensor([[1.0000e+00, 1.0000e+00, ..., 1.3000e+01, 1.6000e+01],
        [2.0000e+00, 1.0000e+00, ..., 3.2000e+01, 4.0000e+01],
        ...,
        [1.7378e+04, 3.1000e+01, ..., 4.8000e+01, 6.1000e+01],
        [1.7379e+04, 3.1000e+01, ..., 3.7000e+01, 4.9000e+01]])
```

Набор данных содержит для каждого часа следующие величины.

- Индекс записи: `instant`.
- День месяца: `day`.
- Время года: `season` (1: весна, 2: лето, 3: осень, 4: зима).
- Год: `yr` (0: 2011, 1: 2012).
- Месяц: `mnth`.
- Час: `hr` (от 0 до 23).
- Признак праздничного дня: `holiday`.
- День недели: `weekday`.
- Признак рабочего дня: `workingday`.
- Погодная ситуация: (1: ясно, 2: туман, 3: небольшой дождь/снег, 4: сильный дождь/снег).
- Температура в градусах Цельсия: `temp`.

- Ощущаемая температура в градусах Цельсия: `atemp`.
- Влажность: `hum`.
- Скорость ветра: `windspeed`.
- Количество временных пользователей: `casual`.
- Количество зарегистрированных пользователей: `registered`.
- Число велосипедов для аренды: `cnt`.

В наборе данных временных рядов наподобие этого строки соответствуют последовательным моментам времени: они упорядочены по одному из измерений. Конечно, можно считать все строки независимыми и пытаться предсказать количество находящихся в обращении велосипедов на основе, скажем, конкретного времени суток, вне зависимости от предыдущих событий. Однако наличие упорядоченности предоставляет нам возможность извлечь выгоду из причинно-следственных связей во времени. Например, позволяет предсказать поездки на велосипедах, исходя из факта прошедшего ранее дождя. Пока что мы сосредоточим свое внимание на преобразовании нашего набора данных по аренде велосипедов в нечто подходящее для подачи на вход нейронной сети порциями фиксированного размера.

Этой нейросетевой модели понадобится несколько последовательностей значений для каждой отдельной величины, например количества поездок, времени суток, температуры и погодных условий: N параллельных последовательностей размера C . C означает канал (channel) в терминологии нейронных сетей, и это то же самое, что и *столбец* в одномерных данных наподобие наших. Измерение N отражает ось времени, в данном случае по одной записи для каждого часа.

4.4.2. Компоновка данных по периоду времени

Возможно, имеет смысл разбить двухгодичный набор данных на более широкие периоды наблюдения, например дни. Таким образом, у нас получится N (от *number of samples* — «количество примеров данных») наборов из C последовательностей длины L . Другими словами, наш набор данных временных рядов будет представлять собой тензор размерности 3 с формой $N \times C \times L$. Число каналов C останется 17, а L будет 24: по одному на каждый час суток. Никакой особой причины для использования порций по 24 часа нет, хотя общий дневной ритм, вероятно, откроет нам закономерности, пригодные для выполнения предсказаний. Все это зависит, естественно, от правильности размера нашего набора данных — количество строк должно быть кратным 24 или 168. Кроме того, по этой же причине во временном ряду не может быть пропусков.

Вернемся к нашему набору данных по аренде велосипедов. Первый столбец — индекс (общая упорядоченность данных), второй — дата, а шестой — время

суток. У нас есть все нужное для создания набора данных дневных последовательностей количества поездок и прочих внешних величин. Наш набор данных уже отсортирован, но если бы это было не так, мы могли бы воспользоваться `torch.sort`, чтобы упорядочить его соответствующим образом.

ПРИМЕЧАНИЕ

В используемую нами версию файла, `hour-fixed.csv`, были добавлены строки, отсутствующие в исходном наборе данных. Мы предположили, что в эти отсутствующие часы количество активных велосипедов было нулевым (обычно речь идет о ранних утренних часах).

Чтобы получить нужный набор почасовых данных, нам достаточно просмотреть тот же тензор батчами по 24 часа. Взглянем на форму и шаги тензора `bikes`:

```
# In[3]:
bikes.shape, bikes.stride()

# Out[3]:
(torch.Size([17520, 17]), (17, 1))
```

Итого 17 520 часов и 17 столбцов. Изменим форму этих данных, чтобы получилось три оси координат: день, час, а затем наши 17 столбцов:

```
# In[4]:
daily_bikes = bikes.view(-1, 24, bikes.shape[1])
daily_bikes.shape, daily_bikes.stride()

# Out[4]:
(torch.Size([730, 24, 17]), (408, 17, 1))
```

Что мы сделали тут? Прежде всего `bikes.shape[1]` равно 17, по количеству столбцов в тензоре `bikes`. Но вся суть этого кода заключается в вызове `view`, который очень важен: он меняет представление тензором тех же самых содержащихся в хранилище данных.

Как вы знаете из предыдущей главы, при вызове метода `view` на тензоре возвращает новый тензор с другими размерностью и шагами без изменения хранилища. Это позволяет перегруппировывать тензор практически без затрат, поскольку никакие данные копировать не нужно. Для вызова `view` нам пришлось указать новую форму возвращаемого тензора. `-1` здесь играет роль «заполнителя», означающего «столько индексов, сколько остается с учетом прочих измерений и исходного количества элементов».

Как вы помните также из предыдущей главы, хранилище — это непрерывный, линейный контейнер для чисел (в данном случае с плавающей запятой). Каждая строка нашего тензора `bikes` сохраняется вслед за предыдущей в соответствующем хранилище. Что подтверждается результатом приведенного выше вызова `bikes.stride()`.

Шаг `daily_bikes` указывает, что продвижение на 1 по измерению часов (второе измерение) требует продвижения на 17 элементов в хранилище (то есть один набор столбцов); а продвижение на 1 по измерению дней (первое измерение) требует продвижения на число элементов, соответствующее длине строки в хранилище, умноженной на 24 (в данном случае $408 = 17 \times 24$).

Как видим, крайнее справа измерение содержит количество столбцов в исходном наборе данных. Далее идет среднее измерение, которое содержит время, разбитое на порции по 24 последовательных часа. Другими словами, получается N последовательностей по L часов в день для C каналов. Для получения желаемого упорядочения $N \times C \times L$ необходимо транспонировать тензор:

```
# In[5]:
daily_bikes = daily_bikes.transpose(1, 2)
daily_bikes.shape, daily_bikes.stride()

# Out[5]:
(torch.Size([730, 17, 24]), (408, 1, 17))
```

Теперь применим к этому набору данных некоторые изученные ранее методики.

4.4.3. Готов для обучения

Величина «погодные условия» — порядковая, с четырьмя уровнями: 1 для хорошей погоды, а 4 — э-э... для очень плохой. Можно считать эту величину категориальной (уровни рассматривать как метки) или непрерывной. Если остановиться на категориальной, необходимо преобразовать эту величину в быстро кодированный вектор и конкатенировать столбцы с набором данных¹.

Чтобы упростить визуализацию данных, мы ограничимся пока что первым днем. Заполним начальными значениями нулевую матрицу с числом строк, равным количеству часов в сутках, и числом столбцов, равным числу уровней погодных условий:

```
# In[6]:
first_day = bikes[:24].long()
weather_onehot = torch.zeros(first_day.shape[0], 4)
```

¹ В некоторых случаях имеет смысл отклониться от проторенного пути. Теоретически можно попробовать отразить описание величины «наподобие категориальной, но с упорядоченностью» непосредственно путем обобщения унитарных кодирований так, чтобы i -й из наших четырех категорий соответствовал вектор, содержащий единицы на позициях $0 \dots i$ и нули на всех остальных. Или аналогично вложениям, которые мы обсудим в подразделе 4.5.4, можно вычислять частичные суммы вложений, при этом имеет смысл делать их положительными. Как и многое на практике, здесь можно попробовать то, что работает для других, систематически экспериментируя с различными вариантами.

```
first_day[:,9]

# Out[6]:
tensor([1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 2, 2,
        2, 2])
```

Теперь заполняем нашу матрицу единицами, в соответствии с уровнем погодных условий в каждой строке. Не забудьте воспользоваться `unsqueeze` для добавления одноэлементного измерения, как мы делали в предыдущем разделе:

```
# In[7]:
weather_onehot.scatter_(
    dim=1,
    index=first_day[:,9].unsqueeze(1).long() - 1,
    value=1.0)
```

← Уменьшаем значения на 1, поскольку погодные условия варьируются от 1 до 4, в то время как индексы начинаются с 0

```
# Out[7]:
tensor([[1., 0., 0., 0.],
        [1., 0., 0., 0.],
        ...,
        [0., 1., 0., 0.],
        [0., 1., 0., 0.]])
```

Наш день начинается с погоды 1 и заканчивается погодой 2, так что, похоже, все правильно.

Наконец, мы производим конкатенацию нашей матрицы с исходным набором данных с помощью функции `cat`. Посмотрим на первые результаты:

```
# In[8]:
torch.cat((bikes[:24], weather_onehot), 1)[:1]

# Out[8]:
tensor([[ 1.0000,  1.0000,  1.0000,  0.0000,  1.0000,  0.0000,  0.0000,
          6.0000,  0.0000,  1.0000,  0.2400,  0.2879,  0.8100,  0.0000,
          3.0000, 13.0000, 16.0000,  1.0000,  0.0000,  0.0000,  0.0000]])
```

Здесь мы произвели конкатенацию исходного набора данных `bikes` и быстро кодированной матрицы «погодных условий» по измерению *столбцов* (то есть 1). Другими словами, столбцы двух наборов данных составляют вместе, то есть к исходному набору данных присоединяются унитарно кодированные столбцы. Для работы функции `cat` необходимы одинаковые размеры тензоров по остальным измерениям: в данном случае по измерению *строк*. Обратите внимание, что последние четыре столбца равны 1, 0, 0, 0 — как раз то, что можно ожидать от признака погодных условий 1.

Можно было сделать то же самое с тензором измененной формы `daily_bikes`. Помните, его форма (B, C, L) , где $L = 24$. Сначала создаем нулевой тензор с теми же B и L , но с числом дополнительных столбцов C :

```
# In[9]:
daily_weather_onehot = torch.zeros(daily_bikes.shape[0], 4,
                                   daily_bikes.shape[2])

daily_weather_onehot.shape

# Out[9]:
torch.Size([730, 4, 24])
```

Теперь заносим быстро кодированные значения в измерение *C* тензора. А поскольку эта операция выполняется там же, меняется только содержимое тензора:

```
# In[10]:
daily_weather_onehot.scatter_(
    1, daily_bikes[:,9,:].long().unsqueeze(1) - 1, 1.0)
daily_weather_onehot.shape

# Out[10]:
torch.Size([730, 4, 24])
```

И производим конкатенацию по измерению *C*:

```
# In[11]:
daily_bikes = torch.cat((daily_bikes, daily_weather_onehot), dim=1)
```

Мы уже упоминали ранее, что это не единственный вариант, как можно поступить с нашей величиной «погодные условия». И действительно, ее метки обладают порядковой связью, так что можно притвориться, что это частные значения непрерывной величины. Просто преобразуем эту величину так, чтобы свести ее область значений к промежутку от 0,0 до 1,0:

```
# In[12]:
daily_bikes[:, 9, :] = (daily_bikes[:, 9, :] - 1.0) / 3.0
```

Как мы упоминали в предыдущем разделе, нормализацию данных к отрезку $[0,0, 1,0]$ или $[-1,0, 1,0]$ желательно производить для всех количественных величин, таких как *temperature* (столбец 10 в нашем наборе данных). Позднее мы увидим, почему так, а пока скажем просто, что это полезно для процесса обучения.

Существует множество вариантов нормализации величин. Можно просто отобразить их на диапазон $[0,0, 1,0]$:

```
# In[13]:
temp = daily_bikes[:, 10, :]
temp_min = torch.min(temp)
temp_max = torch.max(temp)
daily_bikes[:, 10, :] = ((daily_bikes[:, 10, :] - temp_min)
                        / (temp_max - temp_min))
```

или вычесть среднее значение и разделить на стандартное отклонение:

```
# In[14]:
temp = daily_bikes[:, 10, :]
daily_bikes[:, 10, :] = ((daily_bikes[:, 10, :] - torch.mean(temp))
                        / torch.std(temp))
```

Во втором случае среднее значение величины станет равно 0, а стандартное отклонение — 1. При выборке величины из гауссовского распределения 68 % примеров данных попадут в отрезок $[-1, 0, 1, 0]$.

Отлично: мы сформировали еще один прекрасный набор данных и научились работать с данными временных рядов. Для нашего общего обзора важно лишь общее представление о том, как размещаются временные ряды и как предварительно привести данные в форму, пригодную для потребления нейронной сетью.

Прочие наши виды данных напоминают временные ряды наличием строгого упорядочения. Далее в нашем списке следуют текст и аудиоданные. Затем мы разберем текст, а в разделе 4.6 вы найдете ссылки на дополнительные примеры с аудиоданными.

4.5. ПРЕДСТАВЛЕНИЕ ТЕКСТА

Глубокое обучение молниеносно захватило сферу обработки естественного языка (NLP), особенно с использованием моделей, многократно потребляющих сочетание нового входного сигнала и предыдущего выходного сигнала модели. Эти модели называются *рекуррентными нейронными сетями* (*recurrent neural network, RNN*) и с большим успехом применяются для категоризации и генерации текста, а также в системах автоматического перевода. А в последнее время много шума наделали *преобразователи* — класс сетей, позволяющих более гибко учитывать предыдущую информацию. Ранее NLP характеризовалось запутанными многоэтапными конвейерами, включавшими правила, содержащие грамматику языка¹. Теперь же производится комплексное обучение сетей с нуля на больших массивах текста, и эти правила формируются на основе данных. В последние несколько лет именно на глубоком обучении основаны наиболее часто используемые автоматизированные системы перевода, доступные в виде сервисов в интернете.

Наша цель в этом разделе — преобразовать текст в нечто подходящее для обработки нейронной сетью: числовой тензор, точно такой же, как и в предыдущих наших сценариях. Если мы сумеем это сделать, а затем выбрать правильную архитектуру наших заданий обработки текста, то мы будем готовы к обработке естественного

¹ *Nadkarni et al.*, Natural language processing: an introduction, JAMIA, <http://mng.bz/8pJP>. (См. также https://ru.wikipedia.org/wiki/Обработка_естественного_языка.)

языка с помощью PyTorch. Мы сразу же увидим, насколько широкие возможности открываются: с помощью *все тех же инструментов PyTorch* можно достичь производительности на самом современном уровне для множества задач в различных предметных областях; необходимо только привести нашу задачу в нужную форму. Первая часть этой работы заключается в изменении формы данных.

4.5.1. Преобразование текста в числа

Среди уровня работы сети с текстом выделяются два особенно интуитивно понятных: уровень символов, при котором обрабатывается по одному символу за раз, и уровень слов, где наименьшими сущностями с точки зрения сети являются слова. Методика кодирования текстовой информации в виде тензора одна и та же, как при работе на уровне символов, так и на уровне слов. И ничего магического тут нет. Мы уже сталкивались с ней ранее: унитарное кодирование.

Давайте начнем с примера на уровне символов. Прежде всего нам нужен текст для обработки. Прекрасный ресурс для этой цели — проект «Гутенберг» (<http://www.gutenberg.org/>), в рамках которого волонтеры оцифровывают и формируют архив достижений культуры с доступом к нему в открытых форматах, включая обычные текстовые файлы. Если же нам нужно хранилище побольше, то прежде всего стоит обратить внимание на «Википедию»: полное собрание статей «Википедии», содержащее 1,9 миллиарда слов и более 4,4 миллиона статей. На сайте English Corpora можно найти и другие корпуса текста (www.english-corpora.org).

Загрузим «Гордость и предубеждение» Джейн Остин с сайта проекта «Гутенберг»: www.gutenberg.org/files/1342/1342-0.txt. Просто сохраним файл и прочитаем его в code/p1ch4/5_text_jane_austen.ipynb:

```
# In[2]:
with open('../data/p1ch4/jane-austen/1342-0.txt', encoding='utf8') as f:
    text = f.read()
```

4.5.2. One-hot-кодирование символов

Прежде чем продолжить, необходимо позаботиться еще об одном нюансе: кодировке. Это весьма обширная тема, и мы затронем ее лишь вкратце. Для однозначного определения каждого написанного символа мы будем использовать код, представляющий собой последовательность битов соответствующей длины. Простейшей такой кодировкой является ASCII (American Standard Code for Information Interchange, стандартное кодирование США для обмена информацией), созданная еще в 1960-х. ASCII кодирует 128 символов посредством 128 чисел. Например, буква *a* соответствует двоичному числу 1100001 (то есть десятичному 97), буква *b* — двоичному числу 1100010 (то есть десятичному 98) и т. д. Для этой кодировки достаточно 8 бит, что в 1965 году было немалым преимуществом.

ПРИМЕЧАНИЕ

Разумеется, 128 символов недостаточно, чтобы вместить все глифы, буквы с акцентами, лигатуры и все остальное, что необходимо для должного представления написанного текста на прочих языках, кроме английского. Для этой цели было разработано немало кодировок, использующих большее количество битов кода для предсказания более широкого спектра символов. Этот более широкий спектр символов был стандартизирован под названием Unicode, в котором всем известным символам соответствуют числа с битовым представлением, соответствующим конкретной кодировке. В числе наиболее популярных кодировок UTF-8, UTF-16 и UTF-32 с последовательностями 8-, 16- и 32-битных целых чисел соответственно. Строковые значения в Python представляют собой строки символов в кодировке Unicode.

Мы хотим произвести унитарное кодирование имеющихся символов. При этом важно ограничиться набором символов, которые могут потенциально встретиться в анализируемом тексте. В нашем случае, поскольку мы загружаем текст на английском языке, можно спокойно воспользоваться ASCII и работать с небольшой кодировкой. Можно также привести все символы в нижний регистр для сокращения количества различных символов в кодировке. Аналогично этому можно отфильтровать знаки препинания, числа и прочие символы, которые неважны для нужных нам видов текста. В зависимости от решаемой задачи это все может иметь значение для нейронной сети или не иметь.

Сейчас нам нужно произвести синтаксический разбор символов текста и найти унитарное кодирование для каждого из них. Каждому символу необходимо поставить в соответствие вектор длины, равный числу различных символов в кодировке. Этот вектор должен содержать нули везде, за исключением единицы по индексу, соответствующему местоположению символа в кодировке.

Сначала мы разобьем текст на список строк и выберем одну произвольную строку, на которой сосредоточим свое внимание:

```
# In[3]:
lines = text.split('\n')
line = lines[200]
line

# Out[3]:
'"Impossible, Mr. Bennet, impossible, when I am not acquainted with him'
```

Создадим тензор размера, достаточного для хранения всех унитарно кодированных символов для всей строки:

```
# In[4]:
letter_t = torch.zeros(len(line), 128)
letter_t.shape
```

Число 128 жестко зашито в коде
из-за ограничений ASCII

```
# Out[4]:
torch.Size([70, 128])
```

Обратите внимание, что в `letter_t` содержится по одному one-hot-кодированному символу на строку. Нам нужно только установить единицы в нем в соответствующих местах, чтобы все строки отражали нужные символы. Индекс устанавливаемой единицы соответствует индексу символа в кодировке:

```
# In[5]:
for i, letter in enumerate(line.lower().strip()):
    letter_index = ord(letter) if ord(letter) < 128 else 0
    letter_t[i][letter_index] = 1
```

← В тексте используются двойные кавычки, которые не относятся к ASCII, так что мы их здесь отфильтруем

4.5.3. Унитарное кодирование целых слов

Мы унитарно закодировали предложение, получив подходящее для нейронной сети представление. Аналогичным образом можно произвести кодирование на уровне слов, создав словарь, и предложений — последовательностей слов — по строкам нашего тензора. А поскольку словарь содержит много слов, кодированные векторы окажутся очень длинными, что на практике неудобно. Мы увидим в следующем разделе, что существует более эффективный способ представления текста на уровне слов: с помощью *вложений*. А пока продолжим работать с унитарным кодированием и посмотрим, что получится.

Опишем функцию `clean_words`, которая принимает на входе текст и возвращает его в нижнем регистре и без знаков препинания. При вызове ее для строки `Impossible, Mr. Bennet` получается следующее:

```
# In[6]:
def clean_words(input_str):
    punctuation = '.,;:!"?""_-'
    word_list = input_str.lower().replace('\n', ' ').split()
    word_list = [word.strip(punctuation) for word in word_list]
    return word_list

words_in_line = clean_words(line)
line, words_in_line

# Out[6]:
('Impossible, Mr. Bennet, impossible, when I am not acquainted with him',
 ['impossible',
  'mr',
  'bennet',
  'impossible',
  'when',
  'i',
  'am',
  'not',
  'acquainted',
  'with',
  'him'])
```

Далее сформируем соответствие слов индексам в нашей кодировке:

```
# In[7]:
word_list = sorted(set(clean_words(text)))
word2index_dict = {word: i for (i, word) in enumerate(word_list)}

len(word2index_dict), word2index_dict['impossible']

# Out[7]:
(7261, 3394)
```

Обратите внимание, что `word2index_dict` теперь представляет собой ассоциативный массив, в котором слова играют роль ключей, а целые числа — роль значений. С его помощью мы сможем без проблем найти индекс нужного слова при one-hot-кодировании. Теперь сосредоточим внимание на нашем предложении: мы разобьем его на слова и унитарно закодируем: то есть заполним тензор по одному унитарно кодированному вектору на слово. Мы создадим пустой вектор и будем присваивать ему унитарные коды слов в предложении:

```
# In[8]:
word_t = torch.zeros(len(words_in_line), len(word2index_dict))
for i, word in enumerate(words_in_line):
    word_index = word2index_dict[word]
    word_t[i][word_index] = 1
    print('{:2} {:4} {}'.format(i, word_index, word))

print(word_t.shape)

# Out[8]:
0 3394 impossible
1 4305 mr
2 813 bennet
3 3394 impossible
4 7078 when
5 3315 i
6 415 am
7 4436 not
8 239 acquainted
9 7148 with
10 3215 him
torch.Size([11, 7261])
```

Теперь тензор отражает одно предложение длиной 11 в пространстве кодировки размером 7261 (количество слов в нашем словаре). На рис. 4.6 приводится сравнение основных особенностей наших двух способов разбиения текста (и применения вложений, о которых мы поговорим в следующем разделе).

Выбор между кодированием на уровне символов и на уровне слов требует определенного компромисса. Во многих языках символов намного меньше, чем слов: представление символов требует от нас представления лишь нескольких классов, а представление слов — представления колоссального количества классов, при котором на практике все равно придется иметь дело со словами, отсутствующими

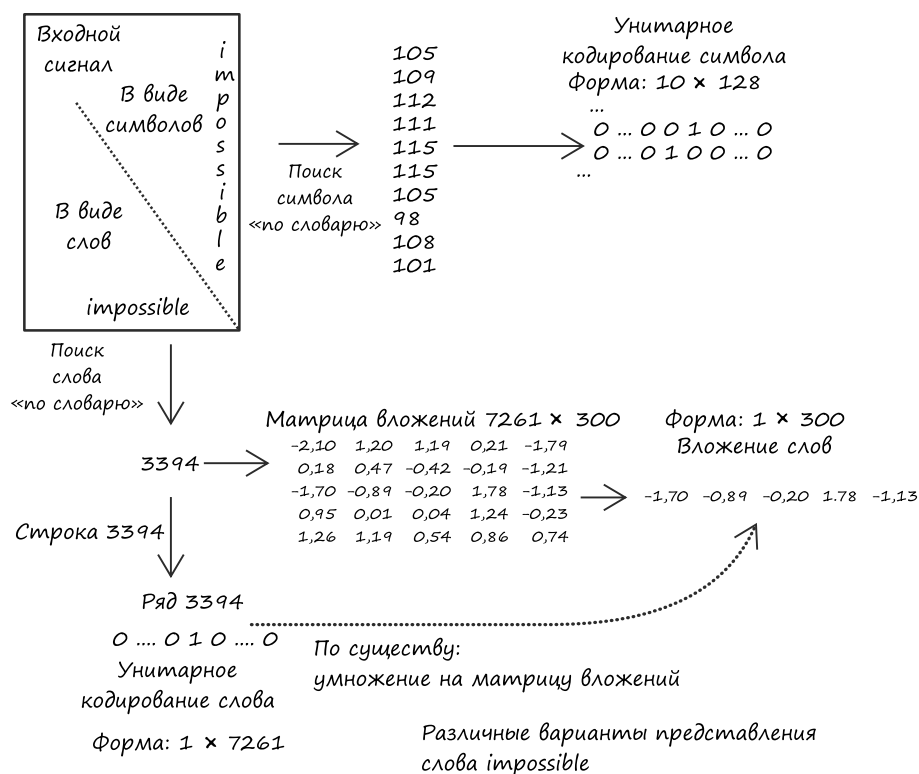


Рис. 4.6. Три способа кодирования слова

в словаре. С другой стороны, слова несут намного больше смысла, чем отдельные буквы, так что представление слов само по себе намного информативнее. Учитывая разительный контраст между этими двумя вариантами, неудивительно, что исследователи искали, находили и с большим успехом применяли промежуточные способы представления: например, метод *кодирования биграмм*¹, который начинается со словаря отдельных символов и к которому затем интерактивно добавляются чаще всего встречающиеся пары букв, пока не будет достигнут заранее заданный размер. При этом предложение из нашего примера можно разбить на следующие токены²:

?Im|pos|s|ible|,|?Mr|.|?B|en|net|,|?impossible|,|?when|?I|?am|?not| ➡
?acquainted|?with|?him

¹ Чаще всего используются реализации из библиотек subword-nmt и SentencePiece. Принципиальный недостаток — представление последовательности символов перестает быть уникальным.

² Взято из токенизатора SentencePiece, обученного на наборе данных для машинного перевода.

В основном тут происходит разбиение по словам. И лишь редкие части — написанное с заглавной буквы слово *Impossible* и имя *Bennet* — состоят из субэлементов.

4.5.4. Вложения текста

Унитарное кодирование — очень удобная методика представления категориальных данных в виде тензоров. Однако, как мы и предвидели, унитарное кодирование перестает работать, когда количество кодируемых элементов, по сути, является неограниченным, как в случае слов в корпусе. В одной книге у нас было более 7000 элементов!

Конечно, мы могли бы провести работу по удалению повторяющихся слов, ограничить использование различных вариантов написания одних слов, а также объединить прошедшее и будущее времена в один токен и т. п. Тем не менее универсальная кодировка английского языка все равно будет *огромной*. Хуже того, всякий раз, когда нам встретится новое слово, придется добавлять в вектор новый столбец, а значит, и добавлять новый набор весовых коэффициентов в модель для этой новой словарной записи — что крайне неудобно с точки зрения обучения.

Как же сжать кодировку до более приемлемого размера и положить конец росту размера? Ну, вместо векторов, состоящих из множества нулей и одной единицы, можно использовать векторы, содержащие числа с плавающей запятой. Вектор, скажем, из 100 чисел с плавающей запятой может представлять действительно большое количество слов. Хитрость в том, чтобы найти эффективный и удобный для дальнейшего обучения способ отображения отдельных слов в это 100-мерное пространство. Это и называется *вложением* (*embedding*).

В принципе, можно просто пройти в цикле по словарю и сгенерировать по набору из 100 случайных чисел с плавающей запятой для каждого слова. Таким образом нам действительно удастся утрамбовать очень большой словарь в набор всего из 100 чисел, но за счет потери какой-либо меры расстояния между словами на основе их смысла или контекста. Использующей подобное вложение слов модели придется довольствоваться очень скудной структурой входных векторов. Идеальным решением было бы сгенерировать вложение таким образом, чтобы используемые в схожем контексте слова отображались на близко расположенные области вложения.

Что ж, при проектировании решения этой задачи вручную мы могли бы сформировать пространство вложения путем отображения основных существительных и прилагательных по осям координат. Можно, например, создать двумерное пространство, в котором оси координат отображаются на существительные *fruit* (0,00–0,33), *flower* (0,33–0,66) и *dog* (0,66–1,00), а также прилагательные *red* (0,00–0,20), *orange* (0,20–0,40), *yellow* (0,40–0,60), *white* (0,60–0,80) и *brown* (0,80–1,00). Наша задача — расположить в пространстве вложений настоящие фрукты, цветы и собак.

При вложении слов можно поставить слову *apple* в соответствие число в квадранте *fruit* и *red*. Аналогично можно отобразить и слова *tangerine*, *lemon*, *lychee* и *kiwi* (в завершение нашего списка цветных фруктов). А затем можно заняться цветами и задать соответствия для *rose*, *poppy*, *daffodil*, *lily* и... кхм... коричневых цветов не так-то много. К *sunflower* подходят *flower*, *yellow* и *brown*, а к *daisy* — *flower*, *white* и *yellow*. Возможно, стоит задать для *kiwi* соответствие поближе к *fruit*, *brown* и *green*¹. Что касается собак и цветов, можно вложить *redbone* недалеко от *red*; кхм... *fox*, наверное, поближе к *orange*; *golden retriever* — к *yellow*, *poodle* — к *white* и... большинство пород собак будет недалеко от *brown*.

Теперь наше вложение выглядит так, как показано на рис. 4.7. И хотя создавать вложения вручную для больших корпусов текста нерационально, обратите внимание, что при размере вложения 2 мы описали 15 различных слов, *помимо восьми основных*, и, вероятно, смогли бы уместить еще несколько, если бы потратили немного времени и приложили воображение.

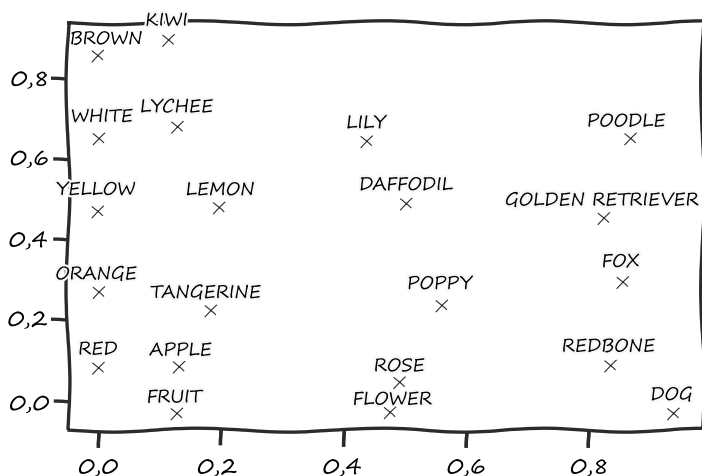


Рис. 4.7. Созданные вручную вложения слов

Как вы, наверное, уже догадались, такую работу можно автоматизировать. Подобные вложения можно генерировать автоматически посредством обработки большого корпуса естественного текста. Основные отличия: в векторе вложений будет от 100 до 1000 элементов, а оси координат не соответствуют непосредственно каким-либо понятиям: просто близкие по смыслу слова отображаются в соседние области пространства вложения, оси которого представляют собой произвольные измерения с плавающей запятой.

¹ На самом деле при нашем одномерном представлении цветов это невозможно, поскольку среднее значение между *yellow* и *brown* у *sunflower* будет *white*, — но основную идею вы поняли, и в пространствах большей размерности она работает лучше.

И хотя конкретные алгоритмы¹ несколько выходят за рамки того, на чем мы хотели бы сосредоточить внимание здесь, хотелось бы упомянуть, что вложения часто генерируются при помощи нейронных сетей, пытающихся предсказать слово в предложении исходя из соседних слов (контекста). В подобном случае можно начать с унитарно кодированных слов и воспользоваться для генерации вложения (обычно довольно неглубокой) нейронной сетью. После генерации вложения его можно использовать для последующих задач в конвейере.

Интересный аспект полученных в результате вложений: близкие слова не только группируются вместе, но их пространственные связи с другими словами также хорошо согласуются друг с другом. Например, если взять вектор вложения для *apple* и начать прибавлять к нему и вычитать из него векторы для других слов, можно получить в результате аналогии вида *apple – red – sweet + yellow + sour* вектор, очень близкий к вектору для слова *lemon*.

Более современные модели вложений — из которых BERT и GPT-2 мелькают в заголовках ведущих СМИ — намного сложнее и чувствительнее к контексту: то есть отображение слова из словаря на вектор не фиксированное, а зависит от окружающего это слово предложения. Тем не менее их часто используют аналогичным с упомянутыми нами классическими вложениями образом.

4.5.5. Вложения текста как схема

Вложения — неотъемлемый инструмент в случаях, когда необходимо представить большое количество записей в словаре с помощью числовых векторов. Но в этой книге мы не станем использовать текст и текстовые вложения, так что вы, наверное, недоумеваете, зачем мы их здесь упомянули. Дело в том, что мы убеждены: представление и обработка текста — прекрасный пример работы с категориальными данными вообще. Вложения полезны во всех случаях, когда one-hot-кодирование становится слишком громоздким. И действительно, в описанной выше форме они представляют собой эффективный способ представления унитарного кодирования, за которым сразу следует умножение на матрицу, в которой векторы вложений играют роль строк.

В не связанных с текстом приложениях обычно нельзя заранее сформировать вложения, но мы начнем со случайных чисел, которых избегали ранее, и подумаем, как с их помощью немного улучшить решение нашей задачи обучения. Это стандартная методика — настолько, что вложения стали заметной альтернативой для one-hot-кодирования любых категориальных данных. С другой стороны, даже при работе с текстом распространенной практикой стало усовершенствование предобученных вложений в ходе решения текущей задачи².

¹ Один из примеров — word2vec: <https://code.google.com/archive/p/word2vec>.

² Методика, называемая точной настройкой (fine-tuning).

Когда нас интересуют случаи совместной встречаемости наблюдаемых слов, вложения слов также могут служить «ориентирами». Например, рекомендательные системы — «покупатели, которым понравилась наша книга, также купили...» — предсказывают, что еще может вызвать интерес покупателя, на основе товаров, которые он уже просматривал, в качестве контекста. Аналогично обработка текста — это, вероятно, самая распространенная, хорошо изученная задача обработки последовательностей; так, например, наработки из сферы обработки естественного языка могут послужить подсказками для решения задач, связанных с временными рядами.

4.6. ИТОГИ ГЛАВЫ

Мы охватили немало материала в этой главе. Мы научились загружать наиболее распространенные типы данных и придавать им вид, подходящий для потребления нейронной сетью. Конечно, в мире существует намного больше форматов данных, чем можно описать в одной книге. Некоторые, например истории болезни, слишком сложны, чтобы тут их описывать. Другие, например аудио- и видеоданные, менее важны для целей этой книги. Впрочем, если вам интересно, мы привели короткие примеры создания аудио- и видеотензоров в дополнительных блокнотах Jupyter, которые вы можете найти на сайте этой книги (<https://www.manning.com/books/deep-learning-with-pytorch>) и в нашем репозитории кода (<https://github.com/deep-learning-with-pytorch/dlwpt-code/tree/master/p1ch4>).

Теперь, когда мы узнали, что такое тензоры и как хранить в них данные, можно сделать следующий шаг к достижению цели нашей книги: узнать, как обучить глубокую нейронную сеть! В следующей главе вас ждет описание внутренней кухни обучения простых линейных моделей.

4.7. УПРАЖНЕНИЯ

1. Сделайте несколько снимков красных, синих и зеленых объектов с помощью вашего телефона или любого цифрового фотоаппарата (или скачайте из интернета, если фотоаппарата под рукой нет).
 - А. Загрузите каждое из изображений и преобразуйте его в тензор.
 - Б. Оцените яркость каждого из изображений с помощью метода `.mean()` соответствующего тензора.
 - В. Вычислите среднее значение каждого из каналов ваших изображений. Можете ли вы распознать красные, синие и зеленые объекты по одному только среднему значению канала?

2. Выберите относительно большой файл с исходным кодом на Python.
 - А. Создайте указатель всех слов в файле исходного кода (можете выбирать степень сложности токенизации по своему усмотрению; мы рекомендуем начать с замены `r"^a-zA-Z0-9_]+"` пробелами).
 - Б. Сравните свой указатель с созданным нами для «Гордости и предубеждения». Какой из них больше?
 - В. Создайте унитарную кодировку для этого файла исходного кода.
 - Г. Какая информация была утрачена при унитарном кодировании? Сравните с потерями при кодировании «Гордости и предубеждения».

4.8. РЕЗЮМЕ

- Для нейронных сетей необходимо, чтобы данные были представлены в виде многомерных числовых тензоров, обычно 32-битных с плавающей запятой.
- В общем случае PyTorch предполагает размещение данных по конкретным измерениям в соответствии с архитектурой модели: например, сверточной или рекуррентной. С помощью API Tensor PyTorch можно легко менять форму данных.
- Совместимость библиотек PyTorch со стандартной библиотекой Python и окружающей ее экосистемой обеспечивает удобство загрузки наиболее распространенных типов данных и преобразование их в тензоры PyTorch.
- Изображения могут включать один или несколько каналов. Наиболее часто встречается набор каналов «красный-зеленый-синий» типичных цифровых фотографий.
- Глубина цвета каждого канала во многих изображениях — 8 бит, хотя встречаются и варианты 12 и 16 бит. Подобную глубину цвета можно хранить в 32-битном числе с плавающей запятой без потери точности.
- В форматах данных с одним каналом явное измерение каналов иногда отсутствует.
- Объемные пространственные данные аналогичны двумерным данным изображений, в них только добавляется третье измерение (глубина).
- Преобразование электронных таблиц в тензоры — достаточно простая задача. Столбцы с категориальными и порядковыми значениями следует обрабатывать иначе, чем столбцы с интервальными.
- Посредством использования словарей можно создать унитарное представление текстовых и категориальных данных. Очень часто вложения можно получить с помощью хороших и эффективных представлений.

5

Внутренняя кухня обучения

В этой главе

- ✓ Как алгоритмы обучаются на данных.
- ✓ Обучение как оценка параметров с помощью дифференцирования и градиентного спуска.
- ✓ Разбор простого алгоритма обучения.
- ✓ Поддержка обучения в PyTorch с помощью компонента autograd.

С расцветом машинного обучения в последнее десятилетие идея машин, усваивающих чужой опыт, стала центральной темой как в технических, так и в журналистских кругах. А теперь давайте разберемся, как может обучаться машина? Какова внутренняя кухня этого процесса или, другими словами, что представляет собой *алгоритм* обучения? С точки зрения наблюдателя, алгоритм обучения представлен входными данными, которым соответствуют желаемые выходные. По завершении обучения алгоритм способен генерировать правильные выходные сигналы при получении новых данных, *достаточно схожих* со входными данными, на которых он обучался. В случае глубокого обучения этот процесс работает даже тогда, когда входные данные и требуемые выходные сигналы *далеки* друг от друга: когда они относятся к различным предметным областям, как, например, изображения и описывающие их фразы, о чем мы говорили в главе 2.

5.1. ВСЕГДА АКТУАЛЬНЫЙ УРОК МОДЕЛИРОВАНИЯ

Построение моделей, позволяющих объяснить взаимосвязь между входными и выходными данными, было начато как минимум несколько веков назад. Иоганн Кеплер (Johannes Kepler), немецкий математик и астроном (1571–1630), открыл в начале 1600-х три закона движения планет на основе данных, собранных его учителем Тихо Браге (Tycho Brahe) в ходе наблюдений невооруженным взглядом (да-да, без помощи каких-либо инструментов и просто записанных на бумаге). Еще до открытия закона всемирного тяготения Ньютона (на самом деле Ньютон как раз и использовал работы Кеплера для открытия своего закона) Кеплер экстраполировал простейшую возможную геометрическую модель, соответствующую полученным данным. И кстати, чтобы сформулировать эти законы, ему потребовалось шесть лет изучать непонятные данные, постепенно осознавая их смысл¹. Этот процесс изображен на рис. 5.1.

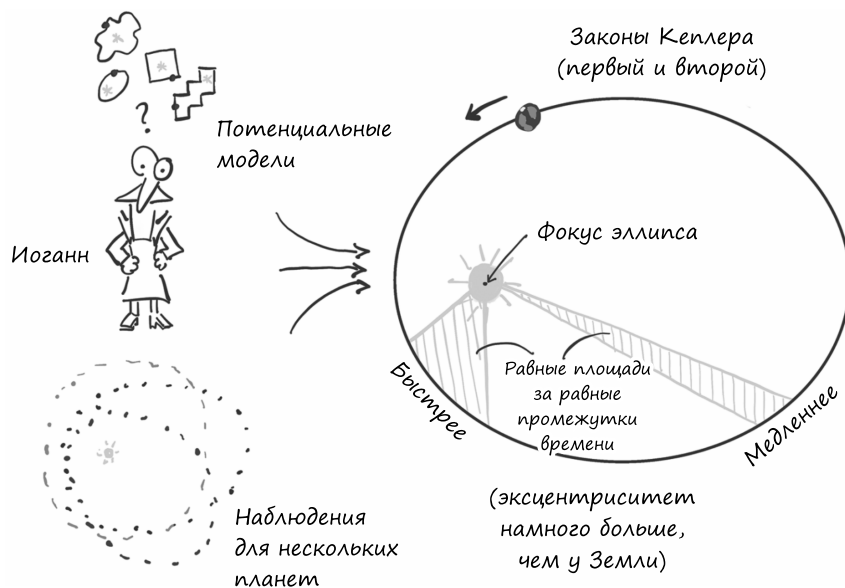


Рис. 5.1. Иоганн Кеплер рассматривает различные потенциальные модели, которые могли бы удовлетворить имеющиеся данные, и в итоге останавливается на эллипсе

Первый закон Кеплера гласит: «Орбита любой планеты представляет собой эллипс, в одном из фокусов которого находится Солнце». Он не знал, почему орбиты являются эллипсами, но по набору наблюдений для конкретной планеты (или спутника крупной планеты, например Юпитера) смог оценить форму

¹ Как рассказывает физик Майкл Фаулер (Michael Fowler): <http://mng.bz/K2Ej>.

(эксцентриситет) и размер (полуфокальный параметр) эллипса. Вычислив по данным эти два параметра, он смог определить, где будет находиться планета во время своего движения по небу. А когда он открыл второй закон — «радиус-вектор, соединяющий планету и Солнце, описывает равные площади за равные промежутки времени», — то смог по имеющимся наблюдениям в разные моменты времени сказать, *когда* планета окажется в конкретной точке пространства¹.

Так как же Кеплер оценил эксцентриситет и размер эллипса без компьютера, карманного калькулятора и даже математического анализа, которые в то время даже не были изобретены? Об этом вспоминает сам Кеплер в своей книге «Новая астрономия» и рассказывает в серии своих статей «Истоки доказательства» Дж. В. Филд (J. V. Field) (<http://mng.bz/9007>):

«По сути, Кеплер перепробовал различные формы, отыскав нужную кривую по определенному количеству наблюдений, а затем на основе найденной кривой определил еще какое-то количество местоположений планет в те моменты времени, для которых у него были наблюдения, и убедился, что вычисленные местоположения соответствуют наблюдаемым».

Дж. В. Филд

Подытожим. За шесть лет Кеплер сделал следующее.

1. Получил много хороших данных от своего друга Браге (не без усилий).
2. Долго пытался их визуализировать, поскольку подозревал, что что-то тут нечисто.
3. Выбрал простейшую возможную модель, которая могла потенциально удовлетворять данным.
4. Разбил данные так, чтобы можно было работать с их частью, выделив независимый набор данных для проверки.
5. Начал с пробных эксцентриситета и размера эллипса и повторял попытки, пока модель не удовлетворила наблюдениям.
6. Проверил модель на независимом наборе наблюдений.
7. Оглянулся на пройденный путь с изумлением.

Типичное руководство по науке о данных издания 1609 года. История науки буквально состоит из этих семи шагов. И за прошедшие столетия мы узнали, что отклоняться от них — верный путь к неудаче².

¹ Для понимания этой главы не обязательно понимать все нюансы законов Кеплера, но прочитать о них подробнее можно тут: https://ru.wikipedia.org/wiki/Законы_Кеплера.

² Если вы, конечно, не физик-теоретик ;).

Именно по этому пути мы и пойдем, чтобы *усвоить* что-то из данных. На самом деле в этой книге нет никакой разницы между терминами «мы будем *подгонять* модель к данным» и «алгоритм будет *обучаться* на данных» («*усваивать* из данных»). В этом процессе всегда используется функция с некоторым количеством неизвестных параметров, значения которых оцениваются исходя из данных: короче говоря, *модель*.

Можно спорить, что *усвоение из данных* предполагает, что используемая модель не предназначена для решения данной конкретной задачи (как эллипс в работах Кеплера), а просто способна аппроксимировать намного более широкое семейство функций. Нейронная сеть прекрасно предсказала бы траектории Тихо Браге без озарения Кеплера, попытавшегося подогнать данные к эллипсу. Впрочем, сэру Исааку Ньютону пришлось бы гораздо больше потрудиться, чтобы вывести законы тяготения из обобщенной модели.

В этой книге нас интересуют модели, не предназначенные для решения конкретной узкой задачи, а способные автоматически адаптироваться к любой из множества схожих задач, на основе пар «входной/выходной сигнал» — другими словами, универсальные модели, обучаемые на данных, относящихся к конкретной решаемой задаче. В частности, PyTorch разработан, чтобы облегчить создание моделей, для которых ошибки при настройке по параметрам можно выразить аналитически. Не волнуйтесь, если это последнее предложение непонятно; далее мы посвятим его разъяснению целый раздел.

Эта глава посвящена автоматизации нахождения универсальных функций. В конце концов, именно это и происходит при глубоком обучении, в котором роль упомянутых универсальных функций играют глубокие нейронные сети, а PyTorch упрощает этот процесс до предела. Чтобы убедиться в правильности понимания всех основных идей, мы начнем с гораздо более простой, чем глубокая нейронная сеть, модели. Это позволит нам разобраться в теоретической внутренней кухне алгоритмов обучения и перейти к более сложным моделям в главе 6.

5.2. ОБУЧЕНИЕ — ЭТО ПРОСТО ОЦЕНКА ПАРАМЕТРОВ

В этом разделе мы расскажем, как выбрать модель для имеющихся данных и оценить ее параметры так, чтобы получить хорошие предсказания для новых данных. Для этого мы попрощаемся с нюансами движения планет и переключимся на вторую по сложности задачу физики: калибровку инструментов.

На рис. 5.2 показана общая картина того, что мы реализуем к концу данной главы. При заданных исходных данных и соответствующих желаемых выходных

сигналах (контрольные данные), а также при начальных значениях весовых коэффициентов модель получает входные данные (прямой проход), после чего вычисляется мера ошибки через сравнение полученных выходных сигналов с контрольными данными. Для оптимизации параметров модели — ее весовых коэффициентов (весов) — вычисляется степень изменения ошибки при единичном изменении весов (то есть *градиент* ошибки относительно параметров) с помощью цепного правила вычисления производной сложной функции (обратный проход). После чего значения весовых коэффициентов обновляются так, чтобы можно было уменьшить ошибку. Процедура повторяется до тех пор, пока оценка ошибки на еще не встречавшихся моделях данных не окажется ниже приемлемого уровня. Если сказанное пока не вполне понятно — у нас есть целая глава, чтобы пояснить, что к чему. К концу главы все части головоломки окажутся на своих местах, а этот абзац станет совершенно понятен.

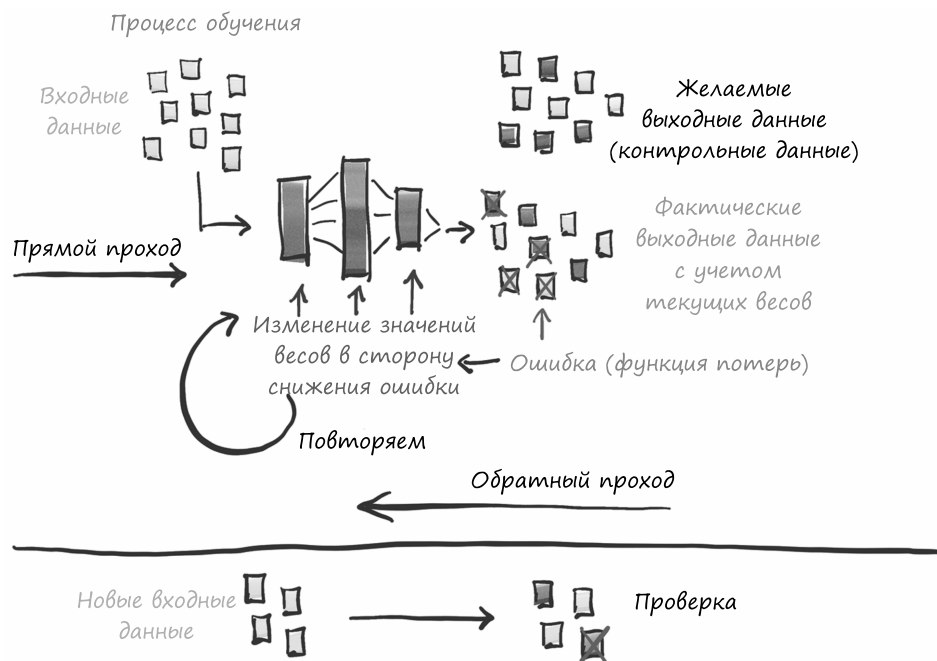


Рис. 5.2. Мысленная модель процесса обучения

Теперь мы возьмем задачу с зашумленным набором данных, создадим модель и реализуем для нее алгоритм обучения. Сначала мы будем делать все вручную, но к концу главы переложим всю тяжелую работу на PyTorch. К концу главы мы охватим многие важнейшие понятия, связанные с обучением глубоких нейронных сетей, хотя наш иллюстрирующий пример очень прост, а модель не является (пока) нейронной сетью.

5.2.1. «Жаркая» задача

Мы вернулись из поездки в жаркие страны и привезли в качестве сувенира элегантный настенный аналоговый термометр. Выглядит он замечательно и прекрасно подходит для гостиной. Единственный недостаток его в том, что не указаны единицы измерения. Не волнуйтесь, у нас есть план: мы создадим набор данных его показаний и соответствующих значений температуры в наших любимых единицах измерения, выберем модель и в цикле будем подбирать значения ее весовых коэффициентов, пока величина ошибки не окажется достаточно низкой, и, наконец, сможем интерпретировать новые показания термометра в понятных нам единицах¹.

Попробуем применить тот же процесс, что и Кеплер. И по ходу дела воспользуемся инструментом, которого у него и в помине не было: PyTorch!

5.2.2. Сбор данных

Начнем с того, что запишем температуру в старых добрых градусах по Цельсию и показания нашего нового термометра. Через несколько недель получаем следующий набор данных (`code/p1ch5/1_parameter_estimation.ipynb`):

```
# In[2]:
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c)
t_u = torch.tensor(t_u)
```

Здесь значения `t_c` — температура в градусах по Цельсию, а значения `t_u` — в неизвестных единицах измерения. В обоих измерениях возможен шум, который появляется как при работе самих приборов, так при снятии нами показаний. Для удобства мы уже поместили данные в тензоры; через минуту мы начнем их использовать.

5.2.3. Визуализация данных

Быстро построив график наших данных на рис. 5.3, мы видим, что они зашумлены, хотя некая закономерность явно прослеживается.

ПРИМЕЧАНИЕ

Внимание, спойлер: мы знаем, что здесь подходит линейная модель, поскольку и задача и данные — вымышленные, но, пожалуйста, наберитесь терпения. На этом наглядном примере мы покажем, что происходит «под капотом» PyTorch.

¹ Подобные задачи — подгонка выходных сигналов модели к непрерывным величинам в контексте обсуждавшихся в главе 4 типов — называются задачами регрессии. В главе 7 и части II мы будем заниматься задачами классификации.

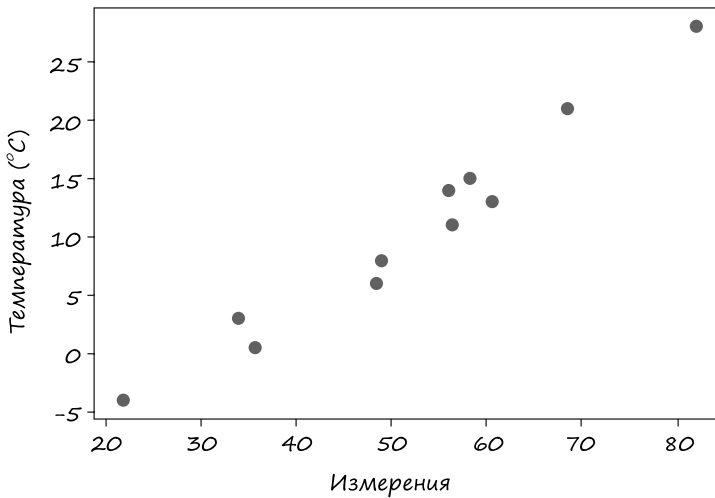


Рис. 5.3. Возможно, наши неизвестные данные описываются линейной моделью

5.2.4. Выбираем линейную модель для первой попытки

Поскольку дополнительной информации у нас нет, предположим, что взаимосвязь двух наборов измерений описывается простейшей возможной моделью, как это мог бы сделать Кеплер. Итак, допустим, что эти наборы связаны линейно, то есть для получения температуры в градусах по Цельсию достаточно умножить t_u на некий коэффициент и прибавить константу (с точностью до ошибки, которую мы не учитываем):

$$t_c = w * t_u + b$$

Обоснованно ли такое предположение? Вероятно; мы увидим далее, насколько хорошие результаты демонстрирует итоговая модель. Мы выбрали названия w и b от слов *вес* (*weight*) и *смещение* (*bias*) — двух очень распространенных терминов для линейного масштабирования и суммируемой постоянной, — мы будем то и дело сталкиваться с ними¹.

Что ж, теперь нам на основе имеющихся данных нужно оценить w и b , параметры нашей модели. Причем так, чтобы температуры, которые мы получим, пропустив через нашу модель неизвестные температуры t_u , оказались близки к фактически измеренным нами температурам в градусах по Цельсию. Если это напоминает вам подбор прямой по серии измерений, то, действительно, именно это мы и делаем.

¹ Весовой коэффициент (вес) указывает, насколько конкретный входной сигнал влияет на выходной. Смещение описывает, каким окажется выходной сигнал, если все входные сигналы будут нулевыми.

Мы воспользуемся PyTorch для разбора этого простого примера и увидим, что обучение нейронной сети, по существу, означает смену модели на чуть более сложную, с несколько (или намного) большим количеством параметров.

Давайте еще конкретизируем: у нас есть модель с неизвестными значениями параметров и нужно получить оценку этих параметров, которая бы минимизировала расхождение между предсказанными выходными сигналами и измеренными значениями (ошибка). Как видим, необходимо дать точное определение меры ошибки. Подобная мера, которую мы будем называть *функцией потерь* (loss function), должна принимать большое значение при высокой ошибке и по возможности как можно более низкое для идеального варианта. Следовательно, целью процесса оптимизации должен быть поиск таких w и b , которые минимизировали бы функцию потерь.

5.3. НАША ЦЕЛЬ — МИНИМИЗАЦИЯ ПОТЕРЬ

Функция потерь (loss function) (она же *функция стоимости* (cost function)) — это функция, возвращающая одно числовое значение, которое процесс обучения должен минимизировать. Вычисление функции потерь обычно означает вычисление разности между желаемыми выходными сигналами для каких-то примеров данных и выходными сигналами, фактически сгенерированными моделью при подаче на ее вход этих примеров. В нашем случае это означает разность между предсказанными нашей моделью температурами t_p и фактически измеренными значениями: $t_p - t_c$.

Необходимо гарантировать, что функция потерь возвращает положительное значение и когда t_p больше, и когда меньше, чем истинное t_c , поскольку наша цель состоит в том, чтобы значение t_p было как можно ближе к значению t_c . Существует несколько вариантов, простейшие из которых $|t_p - t_c|$ и $(t_p - t_c)^2$. В зависимости от выбранного математического выражения можно акцентировать внимание на определенных видах ошибки или, наоборот, сокращать их роль. В теории функция потерь представляет собой способ расстановки приоритетов исправления ошибок, так что обновления параметров приводят к корректировке выходных сигналов для примеров данных с большим весом, а не к изменениям выходных сигналов других примеров данных с меньшим значением функции потерь.

Минимум обеих приведенных функций потерь отчетливо достигается в нуле, они обе монотонно растут по мере удаления предсказанного значения от истинного в любом направлении. Поскольку крутизна графика также монотонно растет по мере удаления от минимума, обе функции — *выпуклые* (convex). Поскольку наша модель линейна, функция потерь (как функция w и b) также будет выпуклой¹.

¹ Сравните с приведенной на рис. 5.6 невыпуклой функцией.

Случаи, когда функция потерь является выпуклой функцией параметров модели, обычно очень удобны, поскольку позволяют очень эффективно находить минимум с помощью специализированных алгоритмов. Однако в этой главе мы воспользуемся более универсальными, хотя и не обладающими столь широкими возможностями, методами. А все потому, что для интересующих нас в конечном итоге нейронных сетей функция потерь не является выпуклой функцией входных сигналов.

Как видно из рис. 5.4, в случае наших двух функций потерь квадраты разностей ведут себя лучше возле минимума: производная функции потерь на основе квадрата ошибки по t_p равна нулю, когда t_p равняется t_c . У модуля же, с другой стороны, производная не определена как раз там, где нам нужна сходимость. На практике это не такая большая проблема, как может показаться, но мы пока что остановим свой взгляд на функции квадрата разности.

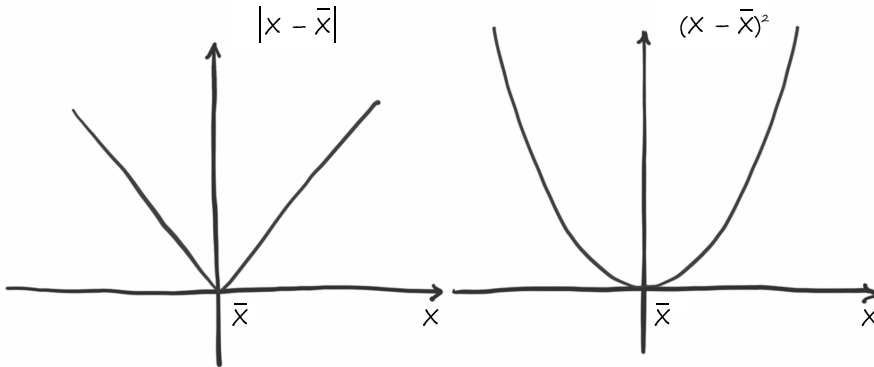


Рис. 5.4. Модуль разности по сравнению с квадратом разности

Стоит отметить, что квадрат разности штрафует далекие от правильных результаты больше, чем модуль разности. Зачастую большое число слегка неправильных результатов лучше, чем небольшое число сильно неправильных, и квадрат разности делает этот вариант приоритетным.

5.3.1. Возвращаемся от задачи к PyTorch

Мы выбрали модель и функцию потерь, то есть большую часть обобщенной картины с рис. 5.2. Осталось запустить процесс обучения и подать на его вход настоящие данные. Нам уже достаточно математической терминологии, поэтому давайте вернемся к PyTorch — в конце концов, мы хотим *получить удовольствие от процесса*. Мы уже создали тензоры данных, а теперь запишем модель в виде функции Python:

```
# In[3]:
def model(t_u, w, b):
    return w * t_u + b
```

Здесь `t_u`, `w` и `b` должны представлять собой входной тензор, весовой параметр и параметр смещения соответственно. В нашей модели параметры представляют собой скалярные значения PyTorch (0-мерные тензоры), а при операции умножения используется транслирование для выдачи возвращаемых тензоров. В любом случае нам пора описать функцию потерь:

```
# In[4]:
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Мы создаем тут тензор разностей, вычисляя их квадраты поэлементно, и наконец получаем скалярную функцию потерь, усредняя все элементы итогового тензора. Такая функция потерь называется *среднеквадратичной* (*mean square loss*).

Теперь можно задать начальные значения параметров, вызвав модель:

```
# In[5]:
w = torch.ones(())
b = torch.zeros(())

t_p = model(t_u, w, b)
t_c

# Out[5]:
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000,
        21.8000, 48.4000, 60.4000, 68.4000])
```

и проверить значение функции потерь:

```
# In[6]:
loss = loss_fn(t_p, t_c)
loss

# Out[6]:
tensor(1763.8846)
```

В этом разделе мы реализовали модель и функцию потерь. И наконец-то добрались до самого интересного в этом примере: как найти такие `w` и `b`, чтобы функция потерь достигала минимума? Сначала мы проделаем это вручную, а потом воспользуемся сверхспособностями PyTorch для решения той же задачи более универсальным, стандартным способом.

ТРАНСЛИРОВАНИЕ

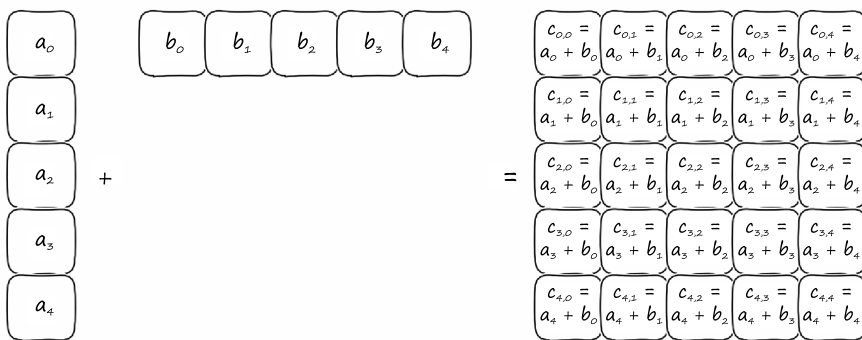
Мы уже упоминали транслирование в главе 3 и обещали обсудить его подробнее, когда оно нам понадобится. В нашем примере есть два скаляра (0-мерных тензора) — `w` и `b`, которые мы перемножаем и складываем с векторами (одномерными тензорами) длиной `b`.

Обычно — и в первых версиях PyTorch тоже — мы можем использовать поэлементные бинарные операции, например сложение, вычитание, умножение и деление, только с аргументами одной формы. Записи по определенным позициям в каждом из тензоров будут использоваться для вычисления соответствующего элемента в тензоре результатов.

Транслирование — очень популярная возможность NumPy, принятая на вооружение в PyTorch, снимает это ограничение для большинства бинарных операций. Подбор соответствующих элементов тензоров производится на основе следующих правил.

- Для каждого индекса измерения, считая с конца, если размер одного из операндов в этом измерении равен 1, PyTorch производит операцию с единственным элементом по этому измерению для каждого из элементов в другом тензоре по этому же измерению.
- Если оба размера превышают 1, они должны совпадать и применяется обычный способ подбора соответствия.
- Если количество индексных измерений у одного из тензоров больше, чем у другого, для каждого элемента по этим измерениям используется весь второй тензор.

Звучит запутанно (и чревато ошибками, если не следить за этим; поэтому мы и назвали измерения тензора так, как показано в разделе 3.4), но обычно можно либо записать измерения тензора и посмотреть, что получится, либо визуализировать транслирование с помощью пространственных измерений, как на следующем рисунке.



Конечно, все это останется чистой теорией, если не привести примеры кода:

```
# In[7]:
x = torch.ones(())
y = torch.ones(3,1)
z = torch.ones(1,3)
```

```

a = torch.ones(2, 1, 1)
print(f"shapes: x: {x.shape}, y: {y.shape}")

print(f" z: {z.shape}, a: {a.shape}")
print("x * y:", (x * y).shape)
print("y * z:", (y * z).shape)
print("y * z * a:", (y * z * a).shape)

# Out[7]:

shapes: x: torch.Size([1]), y: torch.Size([3, 1])
      z: torch.Size([1, 3]), a: torch.Size([2, 1, 1])
x * y: torch.Size([3, 1])

y * z: torch.Size([3, 3])
y * z * a: torch.Size([2, 3, 3])

```

5.4. ВНИЗ ПО ГРАДИЕНТУ

Мы будем оптимизировать функцию потерь относительно параметров с помощью алгоритма *градиентного спуска* (*gradient descent*). В этом разделе мы разберемся, как работает градиентный спуск с самого начала, что очень поможет нам в будущем. Как мы уже упоминали, существуют способы более эффективного решения нашего примера, но эти подходы неприменимы к большинству задач глубокого обучения. На самом деле идея градиентного спуска очень простая и хорошо подходит для больших нейросетевых моделей с миллионами параметров.

Начнем с мысленного представления, схематически изображенного на рис. 5.5. Представьте себе, что вы стоите перед автоматом с двумя ручками с надписями w и b . На экране можно увидеть значение функции потерь, и перед нами теперь поставлена задача это значение минимизировать. Мы не знаем, как ручки влияют на значение функции потерь, и начинаем поворачивать их туда-сюда, чтобы выяснить, поворот в какую сторону каждой из ручек приводит к уменьшению потерь. Мы решаем повернуть обе ручки в стороны, соответствующие уменьшению потерь. Допустим, мы далеки от оптимального значения: вероятно, потери сначала начнут уменьшаться, а затем темпы уменьшения замедлятся по мере приближения к минимуму. В какой-то момент мы замечаем, что потери опять увеличиваются, поэтому решаем повернуть



Рис. 5.5. Карикатура с изображением процесса оптимизации

одну или обе ручки в другую сторону. Мы также выяснили, что при медленном изменении функции потерь имеет смысл поворачивать ручки аккуратнее, чтобы не пропустить момент, когда потери опять начнут расти. Через некоторое время в конце концов процесс сходится к минимуму.

5.4.1. Снижение потерь

Градиентный спуск не так уж сильно отличается от только что описанного сценария. Его идея заключается в вычислении скорости изменения потерь по каждому из параметров и изменении этих параметров в направлении снижения потерь. Подобно тому как мы крутили ручки, можно оценить скорость изменения, прибавив к w и b небольшие значения и посмотрев, насколько меняется функция потерь в этой окрестности:

```
# In[8]:
delta = 0.1

loss_rate_of_change_w = \
    (loss_fn(model(t_u, w + delta, b), t_c) -
     loss_fn(model(t_u, w - delta, b), t_c)) / (2.0 * delta)
```

Этот значит, что в окрестности текущих значений w и b увеличение w на единицу приводит к определенному изменению величины потерь. Если изменение отрицательное, необходимо увеличить w для минимизации потерь, а если положительное — уменьшить. На какую величину? Неплохой идеей будет корректировать w пропорционально скорости изменения потерь, особенно если у функции потерь есть несколько параметров: мы корректируем те из них, которые оказывают существенное влияние на величину потерь. Имеет смысл также менять эти параметры медленно, поскольку скорость изменений может резко отличаться вдали от окрестности текущего значения w . Следовательно, обычно нужно увеличивать скорость изменений лишь на небольшой коэффициент. У этого масштабирующего коэффициента есть много названий; в машинном обучении он называется *скоростью обучения* (*learning rate*):

```
# In[9]:
learning_rate = 1e-2

w = w - learning_rate * loss_rate_of_change_w
```

Делаем то же самое с параметром b :

```
# In[10]:
loss_rate_of_change_b = \
    (loss_fn(model(t_u, w, b + delta), t_c) -
     loss_fn(model(t_u, w, b - delta), t_c)) / (2.0 * delta)

b = b - learning_rate * loss_rate_of_change_b
```

Этот код отражает базовый шаг обновления параметров при градиентном спуске. Если мы будем повторять этот процесс (при условии достаточно маленькой скорости обучения), он постепенно сойдется к оптимальному значению параметров, при котором функция потерь для конкретных данных минимальна. Скоро мы продемонстрируем весь процесс последовательных приближений, но пока что мы довольно грубо вычисляли темп изменений, и поэтому для начала нужно модернизировать наш подход. Давайте посмотрим, как это сделать.

5.4.2. Выражаем аналитически

Вычисление скорости изменения путем последовательных оценок потерь модели, нужное для анализа поведения функции потерь в окрестностях w и b , плохо масштабируется на модели с большим числом параметров. Кроме того, не всегда ясно, насколько большой должна быть эта окрестность. В предыдущем разделе мы выбрали `delta` равной `0.1`, но все зависит от формы графика потерь как функции w и b . Если потери меняются слишком быстро по сравнению с `delta`, будет сложно понять, в каком направлении потери снижаются быстрее всего.

А что, если сделать окрестность бесконечно малой, как на рис. 5.6? Именно это и происходит при аналитическом выражении производной функции потерь относительно какого-либо параметра. В моделях с двумя и более параметрами, как наша, это означает вычисление частных производных функции потерь по каждому из параметров и формирование из них вектора производных: *градиента*.

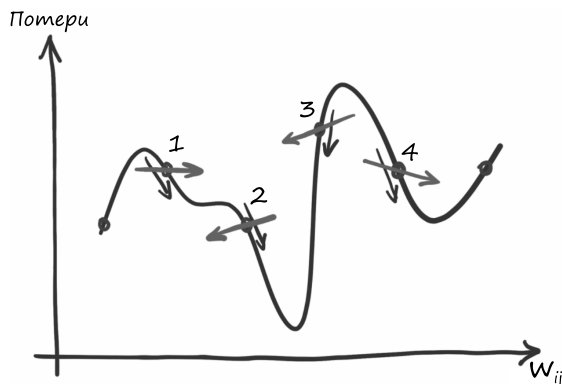


Рис. 5.6. Разница в оценках направлений спуска при вычислении их в дискретных точках и аналитически

Вычисление производных

Для вычисления производной функции потерь по параметру можно использовать цепное правило и вычислить производную функции потерь по ее входному сигналу (то есть выходному сигналу модели), умножив на производную модели по этому параметру:

$$d \text{ loss_fn} / d w = (d \text{ loss_fn} / d t_p) * (d t_p / d w)$$

Напомним, что наша модель — линейная функция, а функция потерь равна сумме квадратов. Выведем выражения для производных. Напомним выражение для функции потерь:

```
# In[4]:
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Вспоминая, что $d x^2 / d x = 2 x$, получаем

```
# In[11]:
def dloss_fn(t_p, t_c):
    dsq_diffs = 2 * (t_p - t_c) / t_p.size(0)
    return dsq_diffs
```

Делитель получается из производной математического ожидания

Применение производных к модели

Поскольку наша модель выглядит следующим образом:

```
# In[3]:
def model(t_u, w, b):
    return w * t_u + b
```

получаем следующие производные:

```
# In[12]:
def dmodel_dw(t_u, w, b):
    return t_u

# In[13]:
def dmodel_db(t_u, w, b):
    return 1.0
```

Определение функции градиента

Собирая все воедино, получаем функцию, возвращающую градиент потерь относительно w и b :

```
# In[14]:
def grad_fn(t_u, t_c, t_p, w, b):
    dloss_dtp = dloss_fn(t_p, t_c)
    dloss_dw = dloss_dtp * dmodel_dw(t_u, w, b)
    dloss_db = dloss_dtp * dmodel_db(t_u, w, b)
    return torch.stack([dloss_dw.sum(), dloss_db.sum()])
```

Суммирование — операция, обратная к транслированию, которое мы неявно производим при применении параметров ко всему вектору входных сигналов модели

Та же идея в математических обозначениях показана на рис. 5.7. Опять же мы производим усреднение (то есть суммируем и делим на константу) по всем точкам данных, получая одну скалярную величину для каждой частной производной функции потерь.

Функция потерь

$$L(m_{w,b}(x))$$

$$\downarrow$$

$$C_{w,b} L = \left(\frac{dL}{dw}, \frac{dL}{dw} \right) = \left(\frac{dL}{dm} \cdot \frac{dm}{dw}, \frac{dL}{dm} \cdot \frac{dm}{dw} \right)$$

↑
↑
↑
↑

Градиент
Частные производные
Модель
Параметры

$m_{w,b}(x)$

Рис. 5.7. Производная функции потерь по весовым коэффициентам

5.4.3. Подгонка модели в цикле

Все готово к оптимизации параметров. Начиная с какого-либо предварительного значения параметра, мы в цикле будем обновлять его в течение фиксированного числа итераций либо до того момента, как w и b перестанут меняться. Существует несколько критериев останова; пока что нас устраивает вариант с фиксированным числом итераций.

Цикл обучения

Раз уж мы с этим столкнулись, давайте введем еще несколько терминов. Отдельная итерация обучения, во время которой обновляются параметры для всех обучающих примеров данных, называется *эпохой*.

Полный цикл обучения выглядит следующим образом (code/p1ch5/1_parameter_estimation.ipynb):

```
# In[15]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        w, b = params

        t_p = model(t_u, w, b)  ← Прямой проход
        loss = loss_fn(t_p, t_c)
        grad = grad_fn(t_u, t_c, t_p, w, b)  ← Обратный проход

        params = params - learning_rate * grad

        print('Epoch %d, Loss %f' % (epoch, float(loss)))  ←
    return params
```

Объем выводимой в этой строке для целей журналирования информации может быть очень велик

Настоящая логика журналирования, выводящая приведенную в этом тексте информацию, несколько сложнее (см. ячейку 16 в том же блокноте: <http://mng.bz/rVB8>), но отличия не важны для понимания основных идей этой главы.

Теперь запустим наш цикл обучения:

```
# In[17]:
training_loop(
    n_epochs = 100,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_u,
    t_c = t_c)

# Out[17]:
Epoch 1, Loss 1763.884644
  Params: tensor([-44.1730, -0.8260])
  Grad: tensor([4517.2969, 82.6000])
Epoch 2, Loss 5802485.500000
  Params: tensor([2568.4014, 45.1637])
  Grad: tensor([-261257.4219, -4598.9712])
Epoch 3, Loss 19408035840.000000
  Params: tensor([-148527.7344, -2616.3933])
  Grad: tensor([15109614.0000, 266155.7188])
...
Epoch 10, Loss 90901154706620645225508955521810432.000000
  Params: tensor([3.2144e+17, 5.6621e+15])
  Grad: tensor([-3.2700e+19, -5.7600e+17])
Epoch 11, Loss inf
  Params: tensor([-1.8590e+19, -3.2746e+17])
  Grad: tensor([1.8912e+21, 3.3313e+19])

tensor([-1.8590e+19, -3.2746e+17])
```

Переобучение

Ой, что произошло? Наш процесс обучения буквально разошелся, и потери стали `inf`. Явный признак того, что обновления `params` слишком велики и их значения начинают колебаться туда-сюда, когда очередное обновление приводит к слишком сильному увеличению, а следующее — корректирует еще больше, чем нужно. Процесс оптимизации неустойчив: он *расходится*, вместо того чтобы сходиться к минимуму. Нам требуются все меньшие обновления `params`, в отличие от показанного на рис. 5.8.

Как же ограничить порядок `learning_rate * grad`? Вроде бы это несложно. Можно просто выбрать меньшее значение `learning_rate`, и действительно, скорость обучения является одной из тех величин, которые обычно меняют, когда обучение идет не так гладко, как хотелось бы¹. Обычно скорости обучения меняют сразу на порядки, так что можно попробовать коэффициент `1e-3` или `1e-4`, что приведет

¹ Это называется настройкой гиперпараметров (hyperparameter tuning). Мы обучаем параметры модели, а гиперпараметры управляют процессом обучения, отсюда и название. Обычно их задают вручную. В частности, они не могут быть составной частью самого процесса оптимизации.

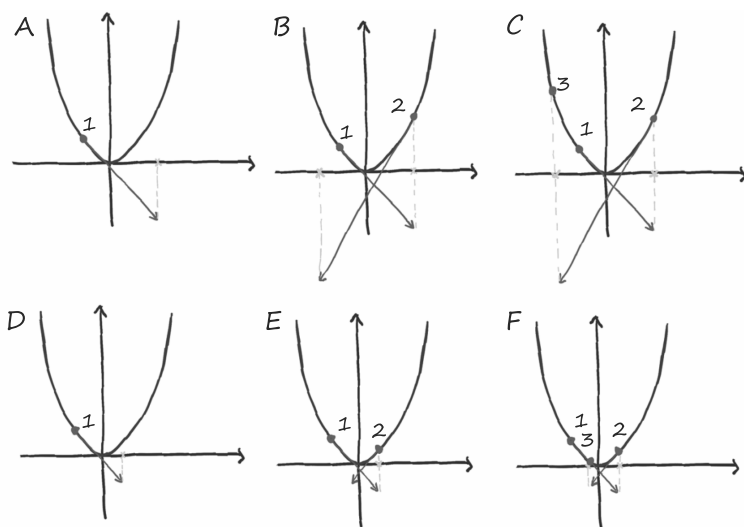


Рис. 5.8. Вверху: процесс оптимизации расходится на выпуклой функции (параболической) из-за слишком больших шагов.

Внизу: процесс оптимизации сходится при маленьких шагах

к снижению величины обновлений сразу на несколько порядков. Возьмем для примера $1e-4$ и посмотрим, что получится:

```
# In[18]:
training_loop(
    n_epochs = 100,
    learning_rate = 1e-4,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_u,
    t_c = t_c)

# Out[18]:
Epoch 1, Loss 1763.884644
  Params: tensor([ 0.5483, -0.0083])
  Grad: tensor([4517.2969, 82.6000])
Epoch 2, Loss 323.090546
  Params: tensor([ 0.3623, -0.0118])
  Grad: tensor([1859.5493, 35.7843])
Epoch 3, Loss 78.929634
  Params: tensor([ 0.2858, -0.0135])
  Grad: tensor([765.4667, 16.5122])
...
Epoch 10, Loss 29.105242
  Params: tensor([ 0.2324, -0.0166])
  Grad: tensor([1.4803, 3.0544])
Epoch 11, Loss 29.104168
  Params: tensor([ 0.2323, -0.0169])
```

```

    Grad: tensor([0.5781, 3.0384])
...
Epoch 99, Loss 29.023582
    Params: tensor([ 0.2327, -0.0435])
    Grad: tensor([-0.0533, 3.0226])
Epoch 100, Loss 29.022669
    Params: tensor([ 0.2327, -0.0438])
    Grad: tensor([-0.0532, 3.0226])

tensor([ 0.2327, -0.0438])

```

Чудесно, теперь процесс стал устойчивым. Но появилась другая проблема: обновления параметров очень малы, так что функция потерь убывает очень медленно и в конце концов вообще замирает. Эту проблему можно устранить посредством адаптации величины `learning_rate`: менять ее в соответствии с порядком обновлений. Есть специально предназначенные для этого схемы оптимизации, одну из которых мы рассмотрим ближе к концу данной главы, в подразделе 5.5.2.

Однако в процессе обновления есть другой потенциальный источник проблем: сам градиент. Вернемся обратно и посмотрим на значение `grad` во время оптимизации на эпохе 1.

5.4.4. Нормализация входных сигналов

Как видим, градиент на первой эпохе для весового коэффициента почти в 50 раз больше, чем градиент для смещения. Это значит, что масштабы весового коэффициента и смещения различны. Если это так, то скорость обучения, достаточно большая для осмысленного обновления одного из них, будет столь велика, что приведет к неустойчивости для другого; а скорость обучения, подходящая для второго, окажется недостаточно велика для осмысленного обновления первого. Это значит, что мы не сможем обновлять параметры, если не поменяем формулировку задачи. Можно использовать отдельные скорости обучения для каждого из параметров, но для моделей с большим числом параметров такое решение создаст слишком много хлопот; так нянчиться с параметрами нам бы не хотелось.

Существует более простой способ держать все под контролем: менять входные сигналы так, чтобы градиенты отличались не сильно. Грубо говоря, можно позаботиться о том, чтобы диапазон входных данных не сильно отличался от диапазона $[-1, 0, 1, 0]$. В нашем случае можно добиться чего-то довольно близкого к этому, просто умножив `t_u` на 0.1:

```

# In[19]:
t_un = 0.1 * t_u

```

Здесь нормализованная версия `t_u` обозначена буквой `n` в конце названия переменной. Теперь можно запустить цикл обучения на нормализованных входных данных:

```
# In[20]:
training_loop(
    n_epochs = 100,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_un,  ← Мы обновили t_u до нашего нового масштабированного t_un
    t_c = t_c)

# Out[20]:
Epoch 1, Loss 80.364342
  Params: tensor([1.7761, 0.1064])
  Grad: tensor([-77.6140, -10.6400])
Epoch 2, Loss 37.574917
  Params: tensor([2.0848, 0.1303])
  Grad: tensor([-30.8623, -2.3864])
Epoch 3, Loss 30.871077
  Params: tensor([2.2094, 0.1217])
  Grad: tensor([-12.4631, 0.8587])
...
Epoch 10, Loss 29.030487
  Params: tensor([ 2.3232, -0.0710])
  Grad: tensor([-0.5355, 2.9295])
Epoch 11, Loss 28.941875
  Params: tensor([ 2.3284, -0.1003])
  Grad: tensor([-0.5240, 2.9264])
...
Epoch 99, Loss 22.214186
  Params: tensor([ 2.7508, -2.4910])
  Grad: tensor([-0.4453, 2.5208])
Epoch 100, Loss 22.148710
  Params: tensor([ 2.7553, -2.5162])
  Grad: tensor([-0.4446, 2.5165])

tensor([ 2.7553, -2.5162])
```

И хотя мы вернули скорости обучения значение `1e-2`, параметры не растут неограниченно во время обновлений в цикле. Взглянем на градиенты: они одного порядка, так что для обоих параметров подходит одна величина `learning_rate`. Наверное, можно лучше нормализовать данные, чем просто масштабировать их в десять раз, но, поскольку для наших нужд этого достаточно, остановимся пока на этом варианте.

ПРИМЕЧАНИЕ

Такой нормализации вполне достаточно, чтобы сеть обучалась, но можно поспорить, что для этой конкретной задачи оптимизировать параметры как раз не обязательно. И это правда! Задача настолько мала, что существует множество способов управиться с параметрами. Однако в более крупных и сложных задачах нормализация — простой и эффективный (а возможно, и неотъемлемый) инструмент улучшения сходимости модели.

Выполним цикл в течение количества итераций, достаточного, чтобы увидеть своими глазами уменьшение изменений `params`. Изменим количество эпох до 5000¹:

```
# In[21]:
params = training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_un,
    t_c = t_c,
    print_params = False)
```

```
params
```

```
# Out[21]:
Epoch 1, Loss 80.364342
Epoch 2, Loss 37.574917
Epoch 3, Loss 30.871077
...
Epoch 10, Loss 29.030487
Epoch 11, Loss 28.941875
...
Epoch 99, Loss 22.214186
Epoch 100, Loss 22.148710
...
Epoch 4000, Loss 2.927680
Epoch 5000, Loss 2.927648
```

```
tensor([ 5.3671, -17.3012])
```

Отлично: наша функция потерь убывает в ходе изменения параметров по направлению уменьшения градиента. Впрочем, она не обязательно доходит до нуля: возможно, число итераций недостаточно для сходимости к нулю или точки данных не располагаются в точности на прямой. Как мы и предвидели, наши измерения не идеально точны либо при снятии показаний произошло их зашумление.

Но взгляните: значения `w` и `b` выглядят поразительно похожими на коэффициенты преобразования градусов по Цельсию в градусы по Фаренгейту (с учетом произведенной ранее нормализации, когда мы умножали входные сигналы на 0,1). Точные значения: `w=5.5556` и `b=-17.7778`. Наш элегантный термометр все время показывал температуру в градусах по Фаренгейту. Америку мы не открыли, просто убедились, что оптимизация на основе градиентного спуска работает!

¹ Авторы опускают описание варианта функции `training_loop` с параметром `print_params`, который вы можете найти в прилагаемом к книге коде. — *Примеч. пер.*

5.4.5. Визуализируем (снова)

Вернемся к тому, что делали в самом начале: построим график наших данных. Нет, правда, это первое, что стоит сделать при исследовании данных. Всегда стройте графики данных:

```
# In[22]:
%matplotlib inline
from matplotlib import pyplot as plt

t_p = model(t_un, *params)

fig = plt.figure(dpi=600)
plt.xlabel("Temperature (°Fahrenheit)")
plt.ylabel("Temperature (°Celsius)")
plt.plot(t_u.numpy(), t_p.detach().numpy())
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
```

Как вы помните, обучение производится для нормализованных неизвестных единиц измерения. Также мы используем распаковку аргументов

Но мы строим график исходных неизвестных значений

В этом коде используется одна из уловок Python — *распаковка аргументов* (*argument unpacking*): выражение `*params` указывает компилятору передавать элементы `params` в виде отдельных аргументов. В Python распаковка аргументов производится обычно для списков и кортежей, но можно ее применять и для тензоров PyTorch, разбиваемых при этом по старшему измерению. Так что `model(t_un, *params)` эквивалентно `model(t_un, params[0], params[1])`.

В результате выполнения этого кода генерируется рис. 5.9. Похоже, наша линейная модель хорошо описывает данные. Также кажется, что наши измерения несколько хаотичны. Так что можно либо заказать в оптике новую пару очков, либо вернуть наш модный термометр в магазин.

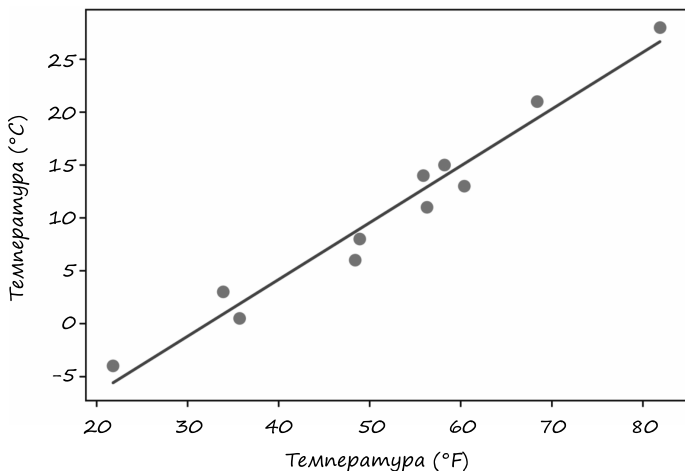


Рис. 5.9. График модели линейной аппроксимации (сплошная линия) и входные данные (кружки)

5.5. КОМПОНЕНТ AUTOGRAD PYTORCH: ОБРАТНОЕ РАСПРОСТРАНЕНИЕ ВСЕГО ЧЕГО УГОДНО

В нашем маленьком путешествии мы только что видели простой пример обратного распространения ошибки: мы вычислили градиент композиции функций — модели и функции потерь — относительно внутренних параметров (w и b) путем распространения производных обратно с помощью *цепного правила*. Основное требование — возможность выражения производной аналитически для всех наших функций. В этом случае можно вычислить градиент — то, что мы ранее называли скоростью изменения потерь, — относительно параметров за раз.

Даже в случае сложной модели с миллионами параметров, если модель дифференцируема, вычисление градиента функции потерь по параметрам сводится к написанию аналитического выражения для производных и вычислению их *один раз*. Правда, работа по написанию аналитического выражения для производных очень глубоко вложенной композиции линейных и нелинейных функций — во все не сахар¹. И времени она занимает немало.

5.5.1. Автоматическое вычисление градиента

Именно тут приходят на помощь тензоры PyTorch благодаря компоненту `autograd` PyTorch. В главе 3 подробно рассказывается, что такое тензоры и какие функции можно для них вызывать. Мы не упомянули, впрочем, один очень интересный нюанс: тензоры PyTorch могут запоминать свою «родословную» в смысле произведенных операций и родительских тензоров и автоматически предоставлять цепочку производных подобных операций относительно их входных сигналов. Это значит, что мы можем не дифференцировать модель вручную² по заданному выражению, неважно, какой степени вложенности, PyTorch автоматически вычисляет его градиент относительно входных параметров.

Использование `autograd`

В текущей ситуации лучше всего переписать наш код калибровки термометра, на этот раз используя автоматическое вычисление градиента, и посмотреть, что получится. Прежде всего напомним нашу модель и функцию потерь (листинг 5.1).

Листинг 5.1. `code/p1ch5/2_autograd.ipynb`

```
# In[3]:
def model(t_u, w, b):
    return w * t_u + b
```

¹ А может и нет; зависит от того, как вы любите проводить выходные!

² Тью! А чем же мы тогда будем заниматься в субботу, а?

```
# In[4]:
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Снова задаем начальные значения тензора параметров:

```
# In[5]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

Атрибут grad

Заметили аргумент `requires_grad` в конструкторе тензора? Он указывает PyTorch отслеживать целое семейство тензоров, получаемых в результате операций над `params`. Другими словами, у любого тензора, произошедшего от `params`, будет доступ к цепочке функций, вызывавшихся для получения из `params` этого тензора. Если эти функции дифференцируемые (как большинство операций над тензорами PyTorch), величина производной будет автоматически занесена в атрибут `grad` тензора `params`.

Вообще говоря, у всех тензоров PyTorch есть атрибут `grad`. Обычно он равен `None`:

```
# In[6]:
params.grad is None

# Out[6]:
True
```

Чтобы заполнить его, достаточно задать аргумент `requires_grad` тензора равным `True`, вызвать модель, вычислить потери, а затем вызвать метод `backward` тензора `loss`:

```
# In[7]:
loss = loss_fn(model(t_u, *params), t_c)
loss.backward()

params.grad

# Out[7]:
tensor([4517.2969, 82.6000])
```

Теперь атрибут `grad` тензора `params` содержит производные функции потерь по всем параметрам.

Когда мы вычисляем `loss`, а параметры `w` и `b` требуют градиентов, помимо выполнения собственно вычислений, PyTorch создает граф автоматического вычисления градиента, в котором роль вершин (показанных черными кружками) играют отдельные операции, как показано в верхнем ряду на рис. 5.10. При вызове `loss.backward()` PyTorch обходит этот граф в обратном порядке, вычисляя градиенты, как показано стрелками в нижнем ряду на рис. 5.10.

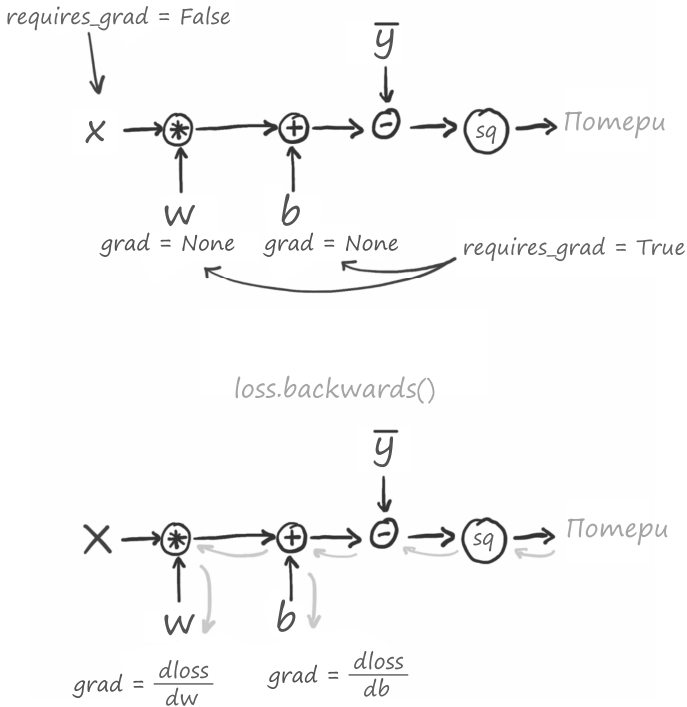


Рис. 5.10. Графы прямого и обратного прохода модели, вычисляемые autograd

Накопление функций grad

Количество тензоров с параметром `requires_grad`, установленным в `True` аргументом, и композиций функций может быть любым. В этом случае PyTorch вычисляет производные функции потерь по всей цепочке функций (графу вычислений) и накапливает их значения в атрибутах `grad` этих тензоров (узлы этого графа).

Внимание! Впереди *большой подводный камень*, о который регулярно спотыкается множество новичков PyTorch и немало более опытных разработчиков тоже. Мы написали выше «*накапливает*», а не «*сохраняет*».

ПРЕДОСТЕРЕЖЕНИЕ

При вызове `backward` производные *накапливаются* в узлах-листьях. Необходимо явным образом *обнулять градиенты* после обновления параметров на их основе.

Повторите вместе с нами: при вызове `backward` производные *накапливаются* в узлах-листьях. Так что, если `backward` вызывался ранее, потери оцениваются опять, `backward` вызывается снова (как и в любом цикле обучения), после чего накапливаются градиенты во всех листьях графа, то есть суммируются

с вычисленными на предыдущей итерации, в результате чего получается неправильное значение градиента.

Чтобы предотвратить подобное, необходимо *явным образом обнулять градиенты* на каждой итерации. Это легко сделать с помощью метода с заменой на месте `zero_`:

```
# In[8]:
if params.grad is not None:
    params.grad.zero_()
```

ПРИМЕЧАНИЕ

Наверное, вы недоумеваете, зачем PyTorch требует обнуления градиентов, а не обнуляет их автоматически при каждом вызове `backward`. Просто этот вариант обеспечивает большую гибкость и возможности контроля при работе с градиентами в сложных моделях.

Хорошенько усвоив это предостережение, мы можем посмотреть, как выглядит код обучения, использующий автоматическое вычисление градиента с начала и до конца:

```
# In[9]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
    return params
```

← Это можно сделать в любой момент, предшествующий вызову `loss.backward()`

← Несколько неуклюжий фрагмент кода, но, как мы увидим в следующем разделе, на практике это не так важно

Обратите внимание, что наш код обновления `params` не так прост, как можно было бы предположить. У него есть две особенности. Во-первых, мы инкапсулируем обновление в контексте `no_grad` с помощью оператора `with` языка Python. Это значит, что механизм автоматического вычисления градиента игнорирует внутренности блока `with`¹: то есть не добавляет ребра в граф прямого прохода. На самом деле при выполнении этого фрагмента кода при вызове `backward` граф прямого прохода, зафиксированный PyTorch, поглощается,

¹ На самом деле он отслеживает, что что-то изменило `params` посредством операции с заменой на месте.

оставляя нас с узлом-листом `params`. Но теперь мы хотели бы изменить этот лист, прежде чем начинать формировать на его основе новый граф прямого прохода. И хотя обычно все это скрыто внутри оптимизаторов, которые мы обсудим в подразделе 5.5.2, мы рассмотрим это внимательнее, когда будем обсуждать еще один распространенный сценарий использования `no_grad` в подразделе 5.5.4.

Во-вторых, мы обновляем `params` с заменой на месте. То есть оставляем тот же самый тензор `params`, просто вычитаем из него обновление. При использовании модуля `autograd` обычно избегают обновлений с заменой на месте, поскольку механизму автоматического вычисления градиента PyTorch могут понадобиться значения, которые бы иначе изменялись для обратного прохода. Здесь же мы работаем без `autograd`, и сохраненный тензор `params` нам может пригодиться. А когда мы будем регистрировать параметры в оптимизаторе в подразделе 5.5.2, будет критически важно не заменять параметры, присваивая соответствующей переменной новые тензоры.

Взглянем на все это в работе:

```
# In[10]:
training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True), ← Добавление
                                                         requires_grad=True
                                                         имеет большое
                                                         значение
    t_u = t_un, ← Опять же мы используем нормализованное t_un вместо t_u
    t_c = t_c)
```

```
# Out[10]:
Epoch 500, Loss 7.860116
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647

tensor([ 5.3671, -17.3012], requires_grad=True)
```

Результат тот же, что и раньше. Отлично! Это значит, что хотя мы *можем* вычислять производные вручную, нам больше не нужно это делать.

5.5.2. Оптимизаторы на выбор

В нашем примере кода мы использовали для оптимизации *простейший* градиентный спуск, вполне достаточный для нашего простого сценария. Разумеется,

существует несколько стратегий и уловок оптимизации, улучшающих сходимость, особенно в случае сложных моделей.

Мы подробнее обсудим этот вопрос в следующих главах, а пока самое время познакомиться со способом абстрагирования PyTorch стратегии оптимизации от пользовательского кода, то есть обсуждавшегося выше цикла обучения. Это избавляет от необходимости писать стереотипный код обновления вручную всех до единого параметров модели. В модуле `torch` есть подмодуль `optim`, в котором можно найти классы, реализующие различные алгоритмы оптимизации. Вот сокращенный их список (`code/p1ch5/3_optimizers.ipynb`):

```
# In[5]:
import torch.optim as optim

dir(optim)

# Out[5]:
['ASGD',
 'Adadelta',
 'Adagrad',
 'Adam',
 'Adamax',
 'LBFGS',
 'Optimizer',
 'RMSprop',
 'Rprop',
 'SGD',
 'SparseAdam',
 ...
]
```

Конструкторы всех оптимизаторов получают в качестве первого аргумента список параметров (тензоров PyTorch с аргументом `requires_grad`, обычно равным `True`). Все передаваемые в оптимизатор параметры хранятся внутри объекта оптимизатора, чтобы оптимизатор мог обновлять их значения и обращаться к их атрибуту `grad`, как показано на рис. 5.11.

У каждого оптимизатора доступны два метода: `zero_grad` и `step`. `zero_grad` обнуляет атрибут `grad` всех передаваемых оптимизатору параметров при его создании. `step` обновляет значения параметров в соответствии с реализуемой конкретным оптимизатором стратегией оптимизации.

Оптимизатор на основе градиентного спуска

Создадим `params` и экземпляр оптимизатора на основе градиентного спуска:

```
# In[6]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-5
optimizer = optim.SGD([params], lr=learning_rate)
```

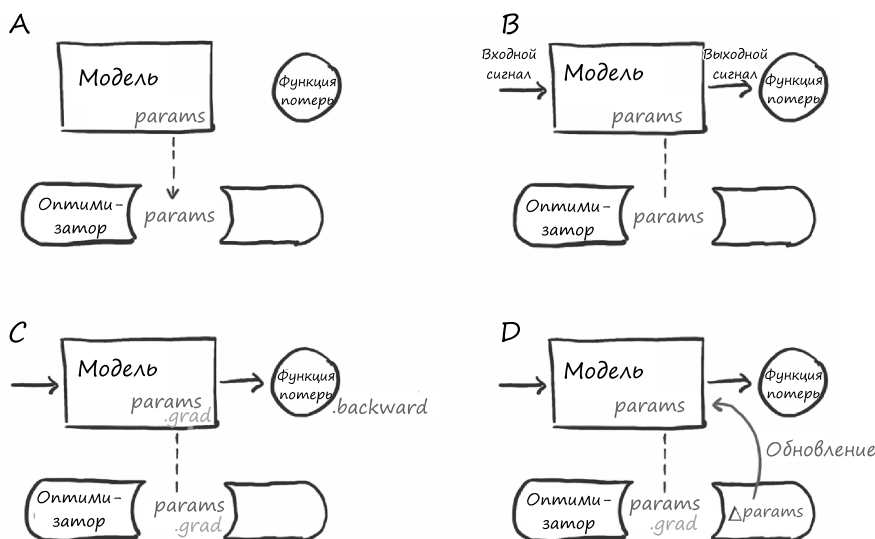


Рис. 5.11. А — схема хранения ссылки на параметры в оптимизаторе. В — после вычисления потерь на основе входных сигналов С вызов `.backward` приводит к заполнению параметров значением `.grad`. D — теперь оптимизатор может обращаться к `grad` и вычислять обновления параметров

Аббревиатура SGD означает *стохастический градиентный спуск* (*stochastic gradient descent*). На самом деле оптимизатор здесь представляет собой простейший градиентный спуск (если аргумент `momentum` равен `0.0` — это значение по умолчанию). *Стохастический*, потому что градиент обычно получается путем усреднения по случайно выбираемому подмножеству всех входных примеров данных, который называется *минибатч* (*minibatch*). Однако оптимизатор не знает, оценивается ли функция потерь на всех примерах данных (простейший градиентный спуск) или на случайном их подмножестве (стохастический), так что алгоритм в обоих случаях совершенно одинаков.

В любом случае испытаем наш новенький крутой оптимизатор в деле:

```
# In[7]:
t_p = model(t_u, *params)
loss = loss_fn(t_p, t_c)
loss.backward()

optimizer.step()

params

# Out[7]:
tensor([ 9.5483e-01, -8.2600e-04], requires_grad=True)
```

Значение `params` обновляется автоматически при вызове `step`, и нам ничего не приходится с ним делать! При этом оптимизатор анализирует `params.grad` и обновляет `params`, вычитая из него произведение `learning_rate` на `grad`, как мы это делали вручную в приведенном выше коде.

Может, самое время вставить этот код в цикл обучения? Нет! Мы чуть не ушибли ногу об упомянутый большой подводный камень: забыли обнулить градиенты. Если бы мы вызывали предыдущий код в цикле, градиенты накапливались бы в листьях при каждом вызове `backward` и наш градиентный спуск пошел бы вкривь и вкось! Вот готовый для цикла обучения код, с дополнительным вызовом `zero_grad` в нужном месте (прямо перед вызовом `backward`):

```
# In[8]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

t_p = model(t_un, *params)
loss = loss_fn(t_p, t_c)
optimizer.zero_grad()
loss.backward()
optimizer.step()

params

# Out[8]:
tensor([1.7761, 0.1064], requires_grad=True)
```

Как и раньше, место вставки этого вызова выбрано в какой-то степени произвольно. Его можно было вставить и ранее в теле цикла

Идеально! Видите, как модуль `optim` помог нам абстрагировать конкретную схему оптимизации? Все, что требуется от нас: указать список параметров (который может оказаться очень длинным, например, в случае очень глубоких нейронных сетей), а обо всех подробностях можно не думать.

Преобразуем наш цикл обучения соответствующим образом:

```
# In[9]:
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```



```
# In[10]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params,
    t_u = t_un,
    t_c = t_c)
```

Важно, чтобы оба параметра были одним и тем же объектом, иначе оптимизатор не будет знать, какие параметры использовались моделью

```
# Out[10]:
Epoch 500, Loss 7.860118
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927680
Epoch 4500, Loss 2.927651
Epoch 5000, Loss 2.927648
```

```
tensor([ 5.3671, -17.3012], requires_grad=True)
```

И снова получаем тот же результат. Замечательно: еще одно подтверждение нашего умения производить градиентный спуск вручную!

Пробуем другие оптимизаторы

Чтобы попробовать другие оптимизаторы, достаточно создать экземпляр нужного оптимизатора, скажем, Adam вместо SGD. Остальной код не меняется. Очень удобно.

Мы не станем углубляться в нюансы использования оптимизатора Adam: достаточно упомянуть, что это более сложный оптимизатор, в котором скорость обучения задается адаптивно. Кроме того, он намного менее чувствителен к масштабу параметров — настолько нечувствителен, что мы можем снова воспользоваться исходным (ненормализованным) входным сигналом `t_u` и даже увеличить скорость обучения до `1e-1`, и Adam даже глазом не моргнет:

```
# In[11]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate)

training_loop(
    n_epochs = 2000,
    optimizer = optimizer,
```

Новый класс оптимизатора

```

params = params,
t_u = t_u,      ← Мы вернулись к исходному t_u в качестве входных данных
t_c = t_c)

```

```

# Out[11]:
Epoch 500, Loss 7.612903
Epoch 1000, Loss 3.086700
Epoch 1500, Loss 2.928578
Epoch 2000, Loss 2.927646

```

```
tensor([ 0.5367, -17.3021], requires_grad=True)
```

Оптимизатор — это не единственная часть нашего цикла обучения, отличающаяся гибкостью. Для обучения другой нейронной сети на тех же данных и с той же функцией потерь достаточно поменять функцию `model`. В данном случае это смысла не имеет, ведь мы знаем, что преобразование градусов по Цельсию в градусы по Фаренгейту сводится к линейному преобразованию, но мы все равно это сделаем в главе 6. Довольно скоро мы увидим, что нейронные сети позволяют отказаться от наших «взятых с потолка» допущений относительно формы аппроксимируемой функции. Более того, мы увидим, как нейронные сети обучаются, даже когда исходные процессы сильно нелинейны (как в случае описания изображения фразой текста из главы 2).

Мы коснулись многих важнейших идей, благодаря которым сможем обучать сложные нейросетевые модели, отчетливо понимая, что происходит у них «за кулисами»: обратное распространение ошибки для оценки градиентов, автоматическое вычисление градиентов и оптимизация весовых коэффициентов моделей с помощью градиентного спуска или других оптимизаторов. На самом деле это не так уж и много. Остается только заполнить пробелы, которые, однако, могут оказаться довольно велики.

Далее вас ждет отступление, посвященное разбиению набора данных, ведь это прекрасный сценарий использования для того, чтобы научиться лучше контролировать автоматическое вычисление градиентов.

5.5.3. Обучение, проверка и переобучение

Иоганн Кеплер научил нас еще кое-чему, что мы пока не обсуждали, помните? Он отделил часть данных, чтобы иметь возможность проверять модели на независимых данных. Это жизненно важно, особенно если используемая модель может потенциально аппроксимировать функции любой формы, как в случае нейронных сетей. Другими словами, очень гибкая модель с большим количеством параметров стремится к минимизации функции потерь *в точках* данных и нет никаких гарантий, что она будет вести себя нужным образом *вдали* или *между* точками данных. В конце концов, именно этого мы и требуем от оптимизатора:

минимизировать функцию потерь ℓ в точках данных. Само собой, при наличии независимых точек данных, не использовавшихся при вычислении потерь или градиентном спуске, мы бы скоро обнаружили, что потери, вычисленные в этих независимых точках данных, выше, чем ожидалось. Мы уже упоминали это явление, называемое *переобучением* (*overfitting*).

Первое, что можно сделать для борьбы с переобучением, — признать наличие такой проблемы. Для этого, как понял Кеплер в 1600-х, необходимо выделить из общего набора несколько точек данных (*проверочный набор данных* (*validation set*)) и обучать модель только на остальных точках (*обучающий набор данных* (*training set*)), как показано на рис. 5.12. Затем при обучении модели можно вычислить функцию потерь один раз на обучающем и один раз — на проверочном наборе данных. Чтобы выяснить, хорошо ли мы подогнали модель к данным, необходимо анализировать оба эти значения!

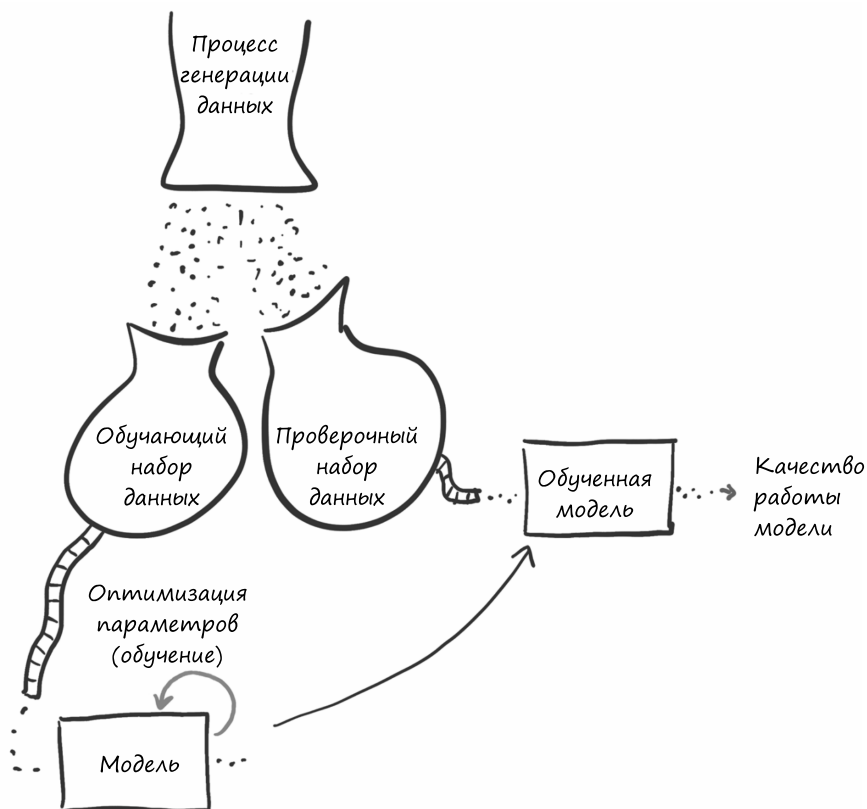


Рис. 5.12. Общая схема процесса генерации данных, а также сбора и использования обучающих данных и независимых проверочных данных

Вычисление потерь на обучающем наборе данных

Потери на обучающем наборе данных показывают, можно ли вообще подогнать нашу модель к этому обучающему набору данных — другими словами, достаточны ли *разрешающие возможности* (*capacity*) этой модели для обработки содержащейся в данных информации. Если бы наш загадочный термометр каким-то образом умудрился измерять температуру по логарифмической шкале, у нашей жалкой линейной модели не было бы ни единого шанса аппроксимировать эти измерения и обеспечить адекватное преобразование в градусы по Цельсию. В подобном случае потери на обучающем наборе данных (значения потерь, которые мы выводили на экран в цикле обучения) перестали бы уменьшаться задолго до достижения нуля.

Глубокая нейронная сеть потенциально может аппроксимировать очень сложные функции при условии достаточно большого числа нейронов, а значит, и параметров. Чем меньше параметров, тем проще должна быть форма функции, чтобы наша сеть смогла ее аппроксимировать. Итак, правило 1: если потери на обучающем наборе данных не уменьшаются, вероятно, модель слишком проста для имеющихся данных. Либо наши данные просто не содержат осмысленной информации, которая позволила бы модели истолковать выходной сигнал: если милая продавщица в магазине продала нам барометр вместо термометра, у нас будет мало шансов предсказать по одному атмосферному давлению температуру в градусах по Цельсию, даже с помощью самой современной архитектуры нейронной сети из Квебека (<https://www.umontreal.ca/en/artificialintelligence/>).

Обобщение на проверочный набор данных

А как насчет проверочного набора данных? Что ж, если вычисленная на проверочном наборе данных функция потерь не убывает вместе с обучающим набором, значит, наша модель обучается лучше аппроксимировать полученные во время обучения примеры данных, но не *обобщается* на примеры данных, которые не входят в этот конкретный набор. Стоит только оценить работу модели на новых, ранее не виденных ею точках, и окажется, что значения функции потерь неудовлетворительны. Итак, правило 2: если потери на обучающем и проверочном наборах данных расходятся — модель переобучена.

Давайте немного углубимся в это явление, вернувшись к нашему примеру с термометром. Мы могли бы попробовать подогнать к данным более сложную функцию, например сплайн или действительно большую нейронную сеть. В результате могла бы получиться модель, блуждающая по точкам данных, как показано на рис. 5.13, просто чтобы функция потерь была близка к нулю. А поскольку поведение функции вне точек данных не увеличивает потери, ничто не держит модель под контролем вдали от обучающих точек данных.

Как решить эту проблему? Хороший вопрос. Из только что сказанного ясно: для решения проблемы переобучения необходимо добиться, чтобы поведение

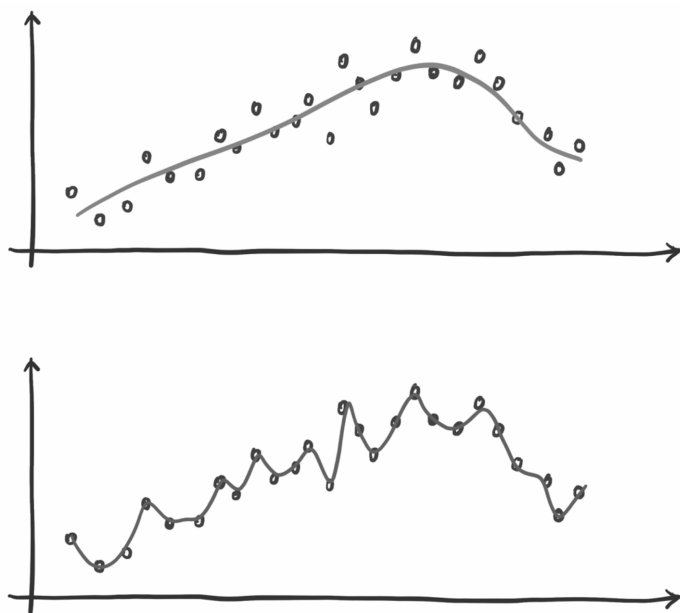


Рис. 5.13. Несколько утрированный пример переобучения

модели *между* точками данных удовлетворяло аппроксимируемому процессу. Прежде всего необходимо убедиться, что у нас достаточно данных, описывающих процесс. Если собирать данные о синусоидальном процессе путем выборки из него через одинаковые интервалы на низкой частоте, подогнать к нему модель будет непросто.

Если допустить, что точек данных у нас достаточно, необходимо гарантировать, что разрешающих возможностей модели достаточно для подгонки ее к обучающим данным между этими точками как можно более равномерно. Есть несколько способов добиться этого. Один из них — добавление в функцию потерь *штрафующих коэффициентов*, чтобы модели стало выгоднее вести себя более гладко и меняться медленнее (до определенной степени). Другой — ввести шум во входные примеры данных, искусственно создавая новые точки данных между обучающими примерами, и заставить модель приспособиться и к ним. Существует еще несколько способов, довольно схожих с описанными. Но лучшая услуга, которую мы можем сделать самим себе, по крайней мере для начала, — упростить модель. С интуитивной точки зрения более простая модель может описывать обучающие данные не так хорошо, как более сложная, но зато, вероятно, будет вести себя более равномерным образом между точками данных.

Налицо необходимость компромисса. С одной стороны, разрешающие возможности модели должны быть достаточны для ее подгонки к обучающему набору

данных. С другой — необходимо избежать переобучения модели. Следовательно, процесс выбора правильного размера нейросетевой модели в смысле количества параметров основан на двух шагах: увеличение размера до тех пор, пока модель не будет хорошо подогнана к данным, а затем уменьшение, пока не будет устранено переобучение.

Больше об этом мы поговорим в главе 12, когда увидим, что вся наша жизнь — балансирование на грани между недообучением и переобучением. А пока вернемся к нашему примеру и посмотрим, как можно разбить данные на обучающий и проверочный наборы данных. Для этого мы перетасуем `t_u` и `t_c` одинаковым образом, а затем разобьем полученные перетасованные тензоры на две части.

Разбиение набора данных

Перетасовка элементов тензора эквивалентна перестановке его индексов — как раз то, что делает функция `randperm`:

```
# In[12]:
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)

shuffled_indices = torch.randperm(n_samples)

train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]

train_indices, val_indices
```

Значения случайные, так что не удивляйтесь, если полученные вами будут отличаться от наших

```
# Out[12]:
(tensor([9, 6, 5, 8, 4, 7, 0, 1, 3]), tensor([ 2, 10]))
```

Теперь можно воспользоваться полученными тензорами индексов для формирования обучающего и проверочного наборов данных на основе исходных тензоров данных:

```
# In[13]:
train_t_u = t_u[train_indices]
train_t_c = t_c[train_indices]

val_t_u = t_u[val_indices]
val_t_c = t_c[val_indices]

train_t_un = 0.1 * train_t_u
val_t_un = 0.1 * val_t_u
```

Цикл обучения особо не меняется. Осталось только дополнительно вычислять потери на проверочном наборе данных на каждой эпохе, чтобы заметить переобучение:

```

# In[14]:
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u,
                  train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        val_t_p = model(val_t_u, *params)
        val_loss = loss_fn(val_t_p, val_t_c)

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
            print(f"Epoch {epoch}, Training loss {train_loss.item():.4f},"
                  f" Validation loss {val_loss.item():.4f}")

    return params

# In[15]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    params = params,
    train_t_u = train_t_un,
    val_t_u = val_t_un,
    train_t_c = train_t_c,
    val_t_c = val_t_c)

# Out[15]:
Epoch 1, Training loss 66.5811, Validation loss 142.3890
Epoch 2, Training loss 38.8626, Validation loss 64.0434
Epoch 3, Training loss 33.3475, Validation loss 39.4590
Epoch 500, Training loss 7.1454, Validation loss 9.1252
Epoch 1000, Training loss 3.5940, Validation loss 5.3110
Epoch 1500, Training loss 3.0942, Validation loss 4.1611
Epoch 2000, Training loss 3.0238, Validation loss 3.7693
Epoch 2500, Training loss 3.0139, Validation loss 3.6279
Epoch 3000, Training loss 3.0125, Validation loss 3.5756

tensor([ 5.1964, -16.7512], requires_grad=True)

```

Эти две пары строк — одинаковы, за исключением train_* и val_* входных сигналов

Обратите внимание на отсутствие тут вызова val_loss.backward(), поскольку мы не хотим обучать модель на проверочном наборе данных

Мы опять используем SGD, поэтому вернулись к нормализованным входным данным

Мы не были полностью справедливы по отношению к нашей модели. Проверочный набор данных очень мал, так что потери на проверочном наборе имеют смысл лишь в известных пределах. В любом случае мы видим, что потери на проверочном наборе данных выше, чем потери на обучающем, хотя и не на порядок. Логично, что модель демонстрирует лучшие результаты на обучающем наборе данных, поскольку именно он определяет параметры модели. Главная

наша цель — убедиться, что убывают как потери на обучающем наборе данных, *так и* потери на проверочном. И хотя значения обеих функций потерь в идеале должны быть примерно одинаковыми, если потери на проверочном наборе данных не отклоняются слишком сильно от потерь на обучающем, значит, наша модель продолжает усваивать обобщенную информацию о наших данных. На рис. 5.14 сценарий С — идеальный, а D — приемлемый. В сценарии А модель вообще не обучается, а в сценарии В наблюдается переобучение. В главе 12 мы увидим более осмысленные примеры переобучения.

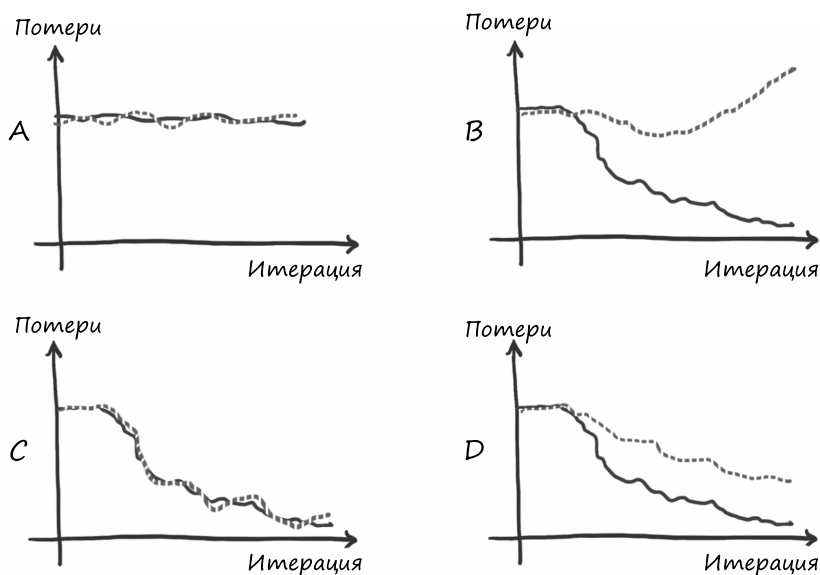


Рис. 5.14. Сценарии переобучения. Сплошная линия обозначает потери на обучающем наборе данных, а штриховая — на проверочном. А — потери на обучающем и проверочном наборах данных не уменьшаются: модель не усваивает ничего, либо по причине того, что в данных не содержится достаточно информации, либо из-за недостаточных разрешающих возможностей модели. В — потери на обучающем наборе данных уменьшаются, а на проверочном — увеличиваются: переобучение. С — потери на обучающем и проверочном наборах данных уменьшаются совершенно синхронно. Поскольку модель еще не достигла грани переобучения, качество ее работы можно улучшить. D — абсолютные значения потерь на обучающем и проверочном наборах данных различны, но общая динамика схожа: переобучение под контролем

5.5.4. Нюансы автоматического вычисления градиентов и его отключение

Из предыдущего цикла обучения видно, что мы вызываем `backward` только для `train_loss`. Следовательно, обратное распространение ошибки происходит

только на основе обучающего набора данных — проверочный набор данных служит для независимой оценки безошибочности выходного сигнала модели на данных, не использовавшихся для обучения.

Любопытный читатель начнет на этом месте задумываться. Модель оценивается дважды — на `train_t_u` и на `val_t_u`, — после чего вызывается `backward`. Не запутается ли вследствие этого механизм автоматического вычисления градиента? Не повлияют ли на `backward` значения, сгенерированные во время прохода по проверочному набору данных?

К счастью, все не так плохо. Первая строка цикла обучения оценивает `model` на `train_t_u`, генерируя `train_t_p`. Затем, исходя из `train_t_p`, вычисляется `train_loss`. Получается граф вычислений, связывающий `train_t_u` с `train_t_p` и с `train_loss`. Когда `model` снова оценивается на `val_t_u`, мы получаем `val_t_p` и `val_loss`. В этом случае создается отдельный граф вычислений, связывающий `val_t_u` с `val_t_p` и с `val_loss`. Через одни и те же функции, `model` и `loss_fn`, пропускаются различные тензоры, что приводит к формированию различных графов вычислений, как показано на рис. 5.15.

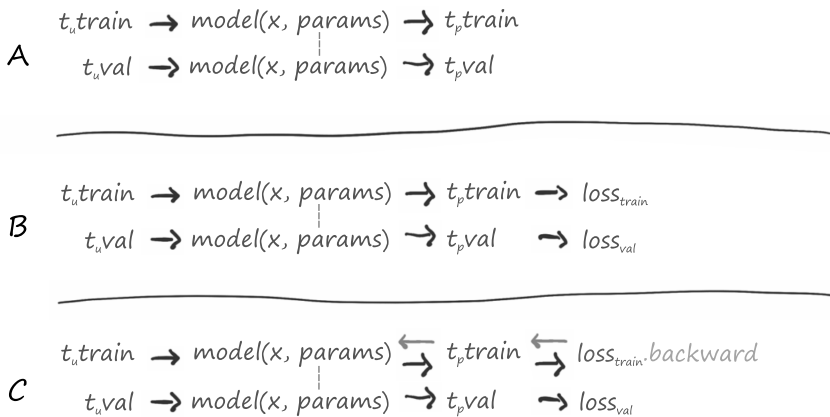


Рис. 5.15. Схема распространения градиентов по графу с двумя величинами потерь при вызове для одной из них `.backward`

Единственные тензоры, общие для этих двух графов, — это параметры. При вызове `backward` для тензора `train_loss` `backward` выполняется для первого графа. Другими словами, мы накапливаем производные `train_loss` относительно параметров, получившихся в результате основанных на `train_t_u` вычислений.

Если же мы (по ошибке) вызовем `backward` и для тензора `val_loss`, то будем накапливать производные `val_loss` относительно параметров *в тех же узлах-листьях*.

Помните, как мы использовали `zero_grad`, когда градиенты накапливались, суммируясь с предыдущими при каждом вызове `backward`, если мы не обнуляли их явным образом? Что ж, здесь могло бы произойти нечто очень похожее: вызов `backward` для `val_loss` привел бы к накоплению градиентов в тензоре `params`, прибавляемых к сгенерированным во время вызова `train_loss.backward()`. В этом случае мы бы фактически обучали нашу модель на всем наборе данных (как обучающем, так и проверочном), ведь градиенты вычислялись бы исходя из обоих.

Еще один любопытный вопрос: раз мы никогда не вызываем `backward` для `val_loss`, зачем вообще формировать граф вычислений? Можно просто вызывать `model` и `loss_fn` как обычные функции, без отслеживания истории вычислений. Несмотря на всю оптимизацию, построение графа автоматического вычисления градиентов означает дополнительные затраты ресурсов, особенно когда модель насчитывает миллионы параметров, которых можно было бы полностью избежать во время прохода проверки.

Для решения этой проблемы PyTorch дает возможность отключать автоматическое вычисление градиентов, когда оно не требуется, с помощью диспетчера контекста `torch.no_grad`¹. На нашей небольшой задаче никаких существенных преимуществ в смысле скорости или объема потребляемой памяти увидеть не получится. Однако в моделях покрупнее различия постепенно накапливаются. Убедиться, что эта возможность работает, можно, посмотрев значение атрибута `requires_grad` тензора `val_loss`:

```
# In[16]:
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u,
                  train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        with torch.no_grad():
            val_t_p = model(val_t_u, *params)
            val_loss = loss_fn(val_t_p, val_t_c)
            assert val_loss.requires_grad == False

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
```

← Диспетчер контекста

Убеждается, что для нашего вывода аргументам `require_grad` принудительно присвоено значение `False` внутри этого блока

¹ Не следует думать, будто использование `torch.no_grad` обязательно означает, что выходные сигналы не требуют градиентов. Существуют отдельные случаи (касающиеся представлений, как обсуждается в подразделе 3.8.1), в которых `requires_grad` не устанавливается в `False` даже при создании в контексте `no_grad`. Для надежности лучше использовать функцию `detach`.

С помощью родственного контекста `set_grad_enabled` можно задать условное выполнение кода с включенным или отключенным `autograd` в соответствии с булевым выражением, обычно указывающим, работает ли код в режиме обучения или в режиме выполнения вывода. Можно, например, описать функцию `calc_forward`, принимающую данные на входе и выполняющую `model` и `loss_fn` с автоматическим вычислением градиентов или без него в соответствии с булевым аргументом `train_is`:

```
# In[17]:
def calc_forward(t_u, t_c, is_train):
    with torch.set_grad_enabled(is_train):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
    return loss
```

5.6. ИТОГИ ГЛАВЫ

Мы начали эту главу с большого вопроса: как машина может обучаться на примерах данных? И до конца главы описывали механизм, с помощью которого можно оптимизировать модель, подогнав ее под данные. Мы выбрали для примера простую модель, чтобы без дополнительных усложнений показать все элементы этого механизма.

Теперь, когда мы уже попробовали аперитив, можем в главе 6 наконец-то приступить к основному блюду: аппроксимации наших данных с помощью нейронной сети. Мы будем решать уже знакомую нам задачу с термометром, но на этот раз с более мощными инструментами, предоставляемыми модулем `torch.nn`. Мы воспользуемся тем же подходом демонстрации на этой маленькой задаче более крупномасштабных способов применения PyTorch. Решение этой задачи не требует нейронной сети, но позволит нам проще разобраться, что необходимо для ее обучения.

5.7. УПРАЖНЕНИЕ

1. Поменяйте модель на $w_2 * t_u ** 2 + w_1 * t_u + b$.
 - А. Какие части цикла обучения и не только необходимо поменять, чтобы учесть подобное новое определение?
 - Б. Какие части безразличны к отключению модуля?
 - В. Выше или ниже стал уровень потерь после обучения?
 - Г. Стал ли фактический результат лучше или хуже?

5.8. РЕЗЮМЕ

- Линейные модели — простейшие приемлемые модели, пригодные для подгонки к данным.
- Для линейных моделей можно использовать методики оптимизации на основе выпуклых функций, но они не обобщаются на нейронные сети, так что мы сосредоточим свое внимание на использовании стохастического градиентного спуска для оценки параметров.
- Для универсальных моделей, не предназначенных для решения конкретной задачи, а автоматически подстраивающихся для решения имеющихся задач, можно применять глубокое обучение.
- Алгоритмы обучения сводятся к оптимизации параметров моделей на основе наблюдений. Функция потерь — мера погрешности выполнения задачи, например расхождения между предсказанными выходными сигналами и измеренными значениями. Цель — минимизировать функцию потерь как можно сильнее.
- Скорость изменения функции потерь относительно параметров модели можно использовать для обновления этих же параметров в направлении убывания потерь.
- Модуль `optim` PyTorch предоставляет набор готовых к использованию оптимизаторов для обновления параметров и минимизации функции потерь.
- Для вычисления градиентов для каждого параметра, в зависимости от того, насколько велик его вклад в итоговый выходной сигнал, оптимизаторы используют модуль `autograd` PyTorch. Благодаря этому пользователи могут применять автоматический граф вычислений во время сложных случаев прямого прохода.
- Для управления поведением модуля `autograd` можно использовать диспетчеры контекста, такие как `with torch.no_grad():`.
- Данные часто разбиваются на отдельные наборы обучающих и проверочных примеров данных, что позволяет оценивать работу модели на данных, на которых она не обучалась.
- Когда качество работы модели все улучшается на обучающем наборе данных, но ухудшается на проверочном, имеет место переобучение. Обычно по причине того, что модель не производит обобщение, а просто запоминает желаемые выходные сигналы для обучающего набора данных.

Аппроксимация данных с помощью нейронной сети

В этой главе

- ✓ Нелинейные функции активации как ключевое отличие от линейных моделей.
- ✓ Модуль nn PyTorch.
- ✓ Решение задачи линейной аппроксимации с помощью нейронной сети.

Пока мы подробно рассмотрели, как обучить линейную модель и как сделать это в PyTorch. Мы сосредоточили свое внимание на очень простой задаче регрессии и использовали линейную модель с одним входным и одним выходным сигналом. Столь простой пример дал нам возможность детально проанализировать внутреннюю кухню обучения модели, не отвлекаясь на реализацию самой этой модели. Как мы видели в обзорной схеме в главе 5, на рис. 5.2 (повторенной на рис. 6.1), для понимания общих процессов обучения модели неважны конкретные нюансы ее устройства. Обратное распространение ошибки на параметры с последующим их обновлением путем вычисления градиента относительно функции потерь не меняется, вне зависимости от сущности модели.

В этой главе мы несколько изменим архитектуру модели: для решения нашей задачи преобразования температуры реализуем полноценную искусственную нейронную сеть. Мы будем и в дальнейшем использовать цикл обучения из прошлой главы, а также разбиение нашего набора примеров преобразования

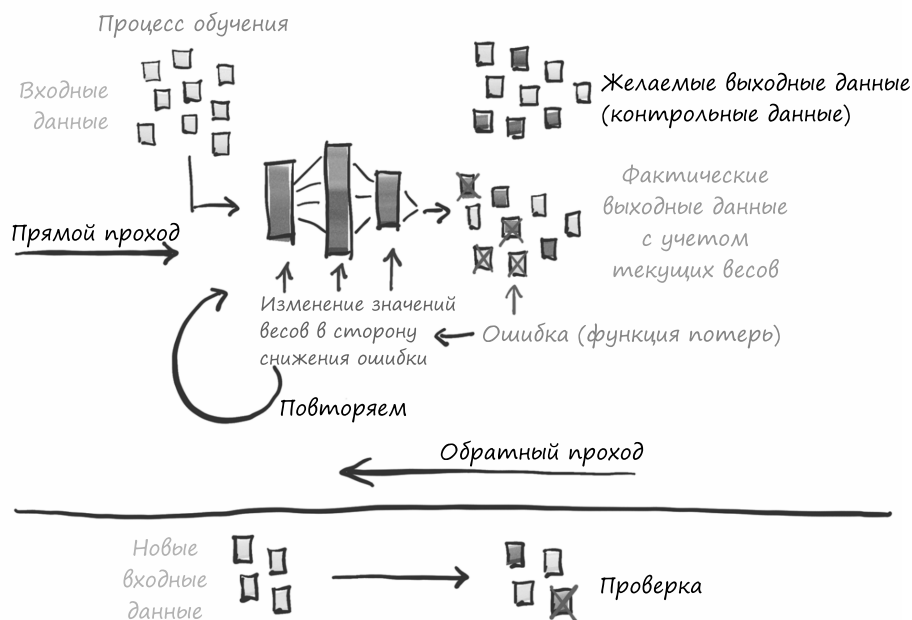


Рис. 6.1. Мысленная модель процесса обучения в реализованном в главе 5 виде

Фаренгейт — Цельсий на обучающий и проверочный наборы. Мы могли бы перейти к использованию квадратичной модели: переписать `model` в виде квадратичной функции входного сигнала (например, $y = a * x^{**2} + b * x + c$). В силу дифференцируемости подобной модели PyTorch позаботился бы о вычислении градиентов, и цикл обучения работал бы как обычно. Интересного в таком варианте, впрочем, для нас немного, ведь мы при этом по-прежнему зафиксировали бы форму функции.

В этой главе мы начнем связывать воедино все результаты подготовительной работы и возможности PyTorch, которые вам предстоит использовать каждый божий день при работе над своими проектами. Вы узнаете, что происходит под крышкой «черного ящика» API PyTorch. Прежде чем заняться реализацией нашей новой модели, обсудим, что имеется в виду под *искусственной нейронной сетью*.

6.1. ИСКУССТВЕННЫЕ НЕЙРОНЫ

В основе глубокого обучения лежит понятие нейронной сети: математической сущности, способной представлять сложные функции путем композиции простых. Термин «*нейронная сеть*» (*neural network*) явно указывает на связь с тем, как работает наш мозг. На самом деле, хотя первые модели и были основаны

на нейронауке¹, современные искусственные нейронные сети лишь отдаленно напоминают механизмы работы нейронов человеческого мозга. Вероятно, как искусственные, так и физиологические нейронные сети используют отдаленно схожие математические стратегии аппроксимации сложных функций, потому что это семейство стратегий очень эффективно.

ПРИМЕЧАНИЕ

Здесь и далее мы отбросим эпитет «искусственный» и будем называть их просто *нейронными сетями*.

Основной «кирпичик» этих сложных функций — *нейрон*, как показано на рис. 6.2. По существу, он представляет собой всего лишь линейное преобразование входного сигнала (например, умножение входного сигнала на какое-либо число (*вес*) и прибавление к нему константы *смещения*) с последующим применением фиксированной нелинейной функции (*функции активации*).

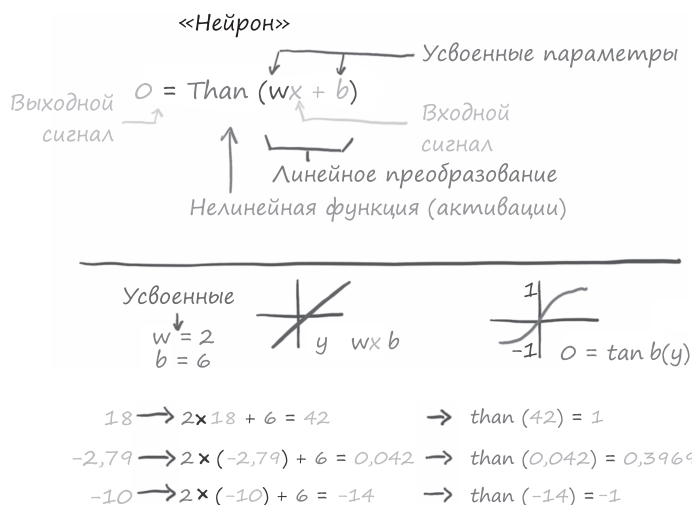


Рис. 6.2. Искусственный нейрон: линейное преобразование, заключенное в нелинейную функцию

Математически можно записать это в виде $o = f(w * x + b)$, где x — входной сигнал, w — вес (коэффициент масштабирования), а b — смещение. f — наша функция активации, в данном случае гиперболический тангенс (функция *tanh*). Вообще говоря, x , а значит, и o могут быть просто скалярами, векторами

¹ См.: Rosenblatt F. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, Psychological Review 65(6), 386–408 (1958), <https://pubmed.ncbi.nlm.nih.gov/13602029/>.

(содержащими много скалярных значений); аналогично w может быть простым скалярным значением или матрицей, а b — скалярным значением или вектором (впрочем, размерность входных сигналов и весовых коэффициентов должна совпадать). В последнем случае приведенное выше выражение называют *слоем* (layer) нейронов, поскольку оно представляет множество нейронов посредством многомерных весовых коэффициентов и смещений.

6.1.1. Формирование многослойной сети

Многослойная нейронная сеть, как показано на рис. 6.3, состоит из композиции функций, подобных тем, которые мы только что описали:

$$\begin{aligned}x_1 &= f(w_0 * x + b_0) \\x_2 &= f(w_1 * x_1 + b_1) \\&\dots \\y &= f(w_n * x_n + b_n),\end{aligned}$$

где выходной сигнал слоя нейронов играет роль входного сигнала для следующего слоя. Учтите, что w_0 здесь — матрица, а x — вектор! Благодаря последнему обстоятельству w_0 может содержать целый *слой* нейронов, а не один весовой коэффициент.

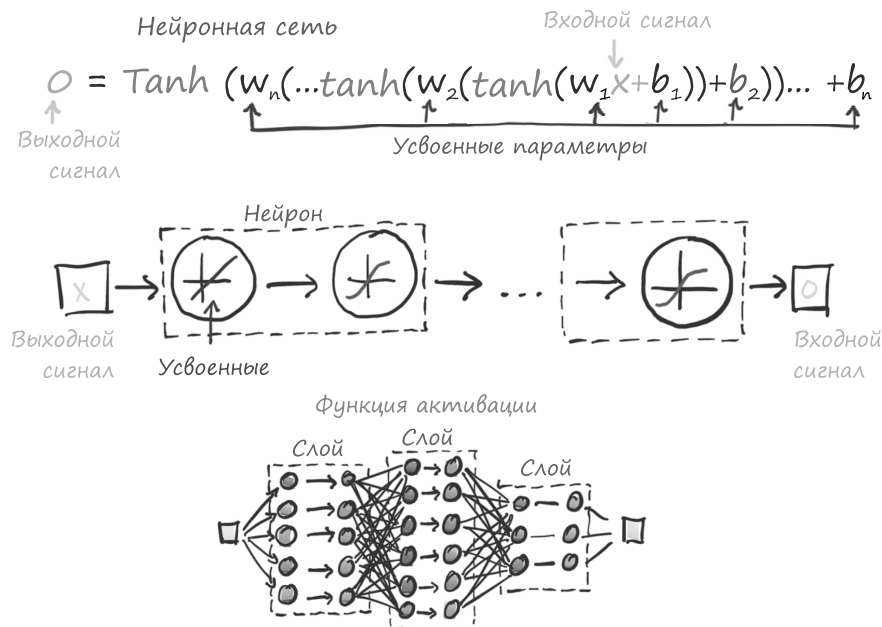


Рис. 6.3. Трехслойная нейронная сеть

6.1.2. Функция ошибки

Важное различие между предыдущей нашей линейной моделью и той, которую мы будем использовать для глубокого обучения, — форма функции ошибки.

Наша линейная модель и функция потерь на основе квадрата ошибки обладали выпуклой кривой ошибки с одним четко определенным минимумом. При использовании других методов можно было искать параметры, автоматически и однозначно минимизирующие функции ошибки. Это значит, что наши обновления параметров пытались *оценить* этот один правильный ответ как можно точнее.

У нейронных сетей отсутствует свойство выпуклости поверхности ошибок даже при использовании той же функции потерь на основе квадрата ошибки! Не существует одного правильного решения для каждого аппроксимируемого параметра. Мы просто пытаемся найти все параметры, которые *совокупно* дали бы подходящий выходной сигнал. А поскольку этот подходящий выходной сигнал лишь *аппроксимирует* истинное значение, он всегда будет слегка неточным.

Эти неточности проявляются несколько произвольным образом, а потому несколько произвольны и параметры, определяющие выходной сигнал (а значит, определяющие и упомянутые неточности). В результате обучение нейронной сети сильно напоминает оценку параметров с технической точки зрения, но теоретическая основа сильно отличается.

Одна из основных причин, из-за которых поверхности ошибок нейронных сетей не выпуклы, связана с функцией активации. Способность набора нейронов аппроксимировать очень широкий спектр полезных функций зависит от присущего отдельным нейронам сочетания линейного и нелинейного поведения.

6.1.3. Все, что нам нужно, — это функция активации

Как мы видели, простейший элемент (глубоких) нейронных сетей — линейное преобразование (масштабирование + смещение) с последующей функцией активации. В нашей предыдущей модели уже было линейное преобразование: вся модель, собственно, представляла собой линейное преобразование. Функция активации играет две важные роли.

- Во внутренних частях модели благодаря функции активации возможны различные наклоны графика выходного сигнала в разных значениях — нечто, по определению недоступное для линейной функции. Искусно сочетая эти по-разному наклоненные участки для различных выходных сигналов,

нейронные сети могут аппроксимировать любые функции, как мы увидим в подразделе 6.1.6¹.

- На последнем слое сети она локализует выходные сигналы предыдущей линейной операции в заданном интервале.

Давайте обсудим, что означает второй из этих пунктов. Пускай мы присваиваем изображениям оценки из разряда «хорошая собачка». Изображения ретриверов и спаниелей получают высокую оценку, а изображения аэропланов и мусоровозов — низкую. Изображения медведей также получают оценку пониже, хотя и выше, чем мусоровозы.

Проблема в том, что надо определиться, что такое высокая оценка: нам доступен весь диапазон `float32`, так что мы можем прийти до очень больших значений. Даже если сказать: «Мы будем использовать 10-балльную шкалу», остается проблема с тем, что модель иногда может выдавать оценку 11 баллов из 10. Как вы помните, «за кулисами» они все представляют собой суммы ($w \cdot x + b$) матричных произведений, которые не ограничены естественным образом никаким диапазоном.

Ограничение диапазона выходных значений

Нам нужно жестко ограничить выходной сигнал нашего линейного преобразования конкретным диапазоном, чтобы получателю этого выходного сигнала не приходилось обрабатывать числовые входные сигналы щенков порядка 12 из 10, медведей порядка -10 и мусоровозов порядка -1000 .

Одна из возможностей — просто обрезать выходные значения: все, что меньше нуля, делать равным 0, все, что больше 10, — равным 10. Соответствующая простая функция активации называется `torch.nn.Hardtanh` (<https://pytorch.org/docs/stable/nn.html#hardtanh>, но учтите, что диапазон по умолчанию — от -1 до 1).

Сжатие выходного диапазона

Еще одно удобное семейство функций — `torch.nn.Sigmoid`, включающее функции $1 / (1 + e^{-x})$, `torch.tanh` и другие, которые мы рассмотрим чуть ниже. Графики этих функций асимптотически стремятся к 0 или -1 при x , стремящемся к минус

¹ Чтобы прочувствовать на интуитивном уровне эти универсальные возможности аппроксимации, можете взять одну из функций активации с рис. 6.5 и сделать из масштабированных (включая умножение на отрицательные числа) и параллельно смещенных ее копий функцию, близкую к нулю почти везде, кроме окрестности $x = 0$, где она больше нуля. При помощи масштабированных, смещенных параллельно и растянутых (по оси X) копий этой базовой функции можно аппроксимировать любую (непрерывную) функцию. Роль такой базовой функции могла бы сыграть, например, функция с рис. 6.6 (справа в среднем ряду). Интерактивный демонстрационный пример можно найти в книге Майкла Нильсена (Michael Nielsen) *Neural Networks and Deep Learning*: <http://mng.bz/Mdon>.

бесконечности, стремятся к 1 при росте x и отличаются более или менее постоянным наклоном при $x=0$. Функции с графиками подобной формы хорошо работают, поскольку нейрон (являющийся опять же просто линейной функцией с последующей функцией активации) оказывается чувствителен к области в центре выходного диапазона нашей линейной функции, а все остальное сосредотачивается поближе к граничным значениям. Как мы видим на рис. 6.4, мусоровоз получит оценку $-0,97$, а медведи/волки/лисы попадут в диапазон от $-0,3$ до $0,3$.

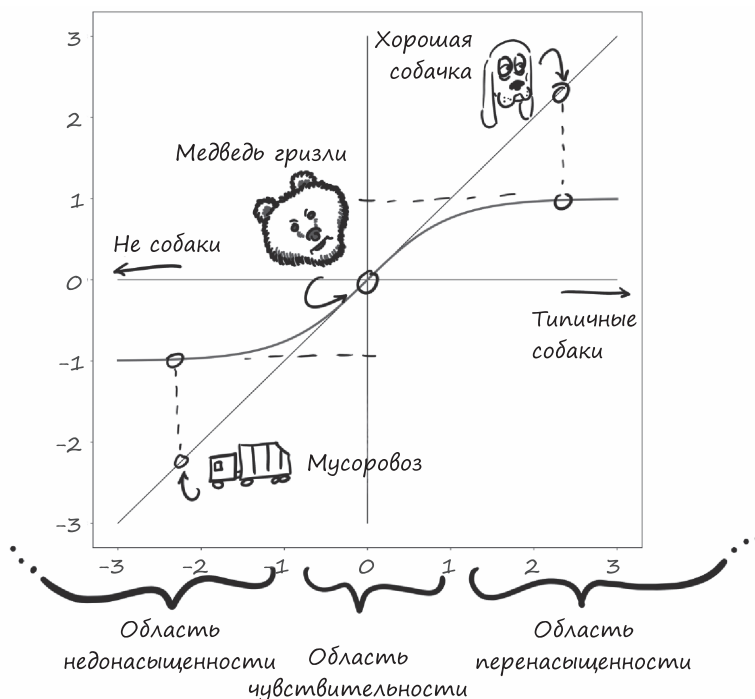


Рис. 6.4. Собаки, медведи и мусоровозы распределяются по степени собакоподобности с точки зрения функции активации \tanh

В результате мусоровозы помечаются как «не собаки», наша хорошая собачка оказывается «явно собакой», а медведь оказывается примерно посередине. Конкретные значения можно увидеть в коде:

```
>>> import math
>>> math.tanh(-2.2)  ← Мусоровоз
-0.9757431300314515
>>> math.tanh(0.1)  ← Медведь
0.09966799462495582
>>> math.tanh(2.5)  ← Хорошая собачка
0.9866142981514303
```

Поскольку медведь находится в диапазоне чувствительности, его небольшие изменения приведут к заметным изменениям результатов. Например, если мы возьмем вместо гризли полярного медведя (у которого несколько более собакоподобная морда), то увидим скачок по оси Y при перемещении в сторону «типичная собака» на графике. И наоборот, коалу модель сочтет менее похожим на собаку, и можно будет наблюдать падение активированного выходного сигнала. Мусоровоз, впрочем, вряд ли можно сделать более похожим на собаку, даже при самых коренных изменениях, выходной сигнал изменится лишь с $-0,97$ до $0,8$ или около того.

6.1.4. Другие функции активации

Существует довольно много функций активации, некоторые из них показаны на рис. 6.5. В первом столбце приведены гладкие функции Tanh и Softplus , а во втором — более «строгие» версии функций активации слева: Hardtanh и ReLU . Особо следует отметить ReLU (*rectified linear unit*, выпрямленный линейный блок), считающийся сейчас одной из лучших универсальных функций активации и используемый во многих современных системах. Функция активации Sigmoid , известная также под названием *логистической функции*, широко применялась

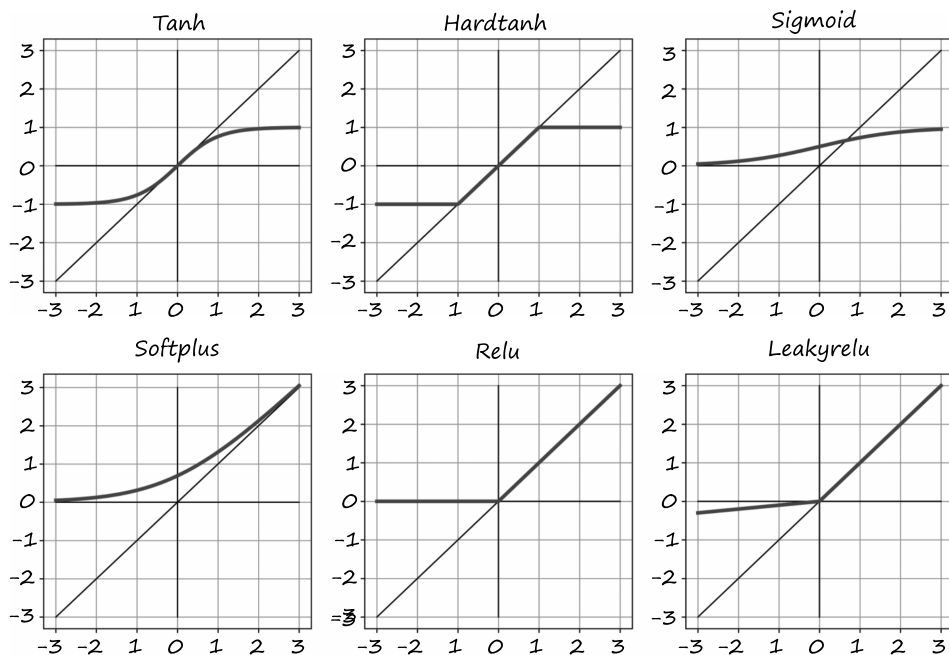


Рис. 6.5. Набор распространенных и не очень распространенных функций активации

в ранних работах по глубокому обучению, но потом перестала широко использоваться, за исключением случаев, когда необходимо явным образом перейти к интервалу $0 \dots 1$: например, когда выходной сигнал должен отражать вероятность. Наконец, функция **LeakyReLU** представляет собой модификацию стандартного **ReLU** с небольшим положительным наклоном графика вместо ровно нуля для отрицательных входных сигналов (обычно этот наклон равен 0,01, но здесь мы показали 0,1 для большей ясности).

6.1.5. Выбор наилучшей функции активации

Несмотря на наличие такого широкого спектра прекрасно работающих функций активации (намного большего, чем показано на рис. 6.5), они интересны практически полным отсутствием жестких требований. Тем не менее функции активации по определению¹:

- нелинейны — сколько ни применяй преобразование вида $(w \cdot x + b)$ без функции активации, все равно получится функция той же самой (аффинной линейной) формы. Нелинейность позволяет сети в целом аппроксимировать более сложные функции;
- дифференцируемы, что дает возможность вычисления градиентов. Точечные разрывы, как мы увидим в **Hardtanh** и **ReLU**, допустимы.

В отсутствие этих характеристик сеть либо превратится обратно в линейную модель, либо с трудом будет поддаваться обучению.

Для функций активации справедливо следующее.

- Имеется по крайней мере один диапазон чувствительности, внутри которого нетривиальные изменения входного сигнала приводят к соответствующим нетривиальным изменениям выходного. Необходимо для обучения.
- У многих из них есть также диапазон нечувствительности (насыщения), в котором изменения входного сигнала практически не приводят к изменениям выходного.

В качестве примера функцию **Hardtanh** можно легко использовать для кусочно-линейной аппроксимации функции с помощью сочетания диапазона чувствительности и применения различных весовых коэффициентов и смещений ко входному сигналу.

Зачастую (хотя далеко не всегда) функции активации отличаются также следующими свойствами:

¹ Конечно, даже эти утверждения не всегда справедливы: см.: *Foerster J. Nonlinear Computation in Deep Linear Networks*, OpenAI, 2019, <http://mng.bz/gygE>.

- наличием нижней границы, к которой функция стремится (или до которой доходит) по мере стремления входного сигнала к минус бесконечности;
- наличием аналогичной верхней границы для плюс бесконечности.

Вспоминая, что нам известно про обратное распространение ошибки, можно догадаться, что ошибки будут распространяться через функцию активации лучше, если входные сигналы находятся в диапазоне чувствительности, мало влияя на нейроны, для которых входной сигнал находится в диапазоне насыщения (поскольку градиент будет близок к нулю из-за плоской части графика около выходного сигнала).

В целом получается механизм, обладающий большими возможностями: при получении на входе различных данных в сети, составленной из линейных и активационных блоков, а) различные нейроны могут возвращать для одних входных сигналов результаты, относящиеся к различным диапазонам, и б) соответствующие этим входным сигналам ошибки в основном влияют на нейроны, работающие в диапазоне чувствительности, а на остальные блоки процесс обучения практически не влияет. Кроме того, поскольку производные функции активации по входным сигналам зачастую близки к 1 в диапазоне чувствительности, оценка параметров линейного преобразования посредством градиентного спуска для работающих в этом диапазоне блоков очень сильно напоминает уже встречавшуюся нам линейную аппроксимацию.

Мы начинаем глубже понимать, почему в результате объединения множества линейных и активационных блоков параллельно и последовательно получается математический объект, способный аппроксимировать сложные функции. Различные сочетания нейронов реагируют в различных диапазонах на входные сигналы, причем параметры этих блоков можно довольно легко оптимизировать посредством градиентного спуска, поскольку процесс обучения напоминает обучение линейной функции, вплоть до момента насыщения выходного сигнала.

6.1.6. Что обучение дает нейронной сети

Модели, создаваемые из наборов линейных преобразований, за которыми следуют дифференцируемые функции активации, потенциально способны на аппроксимацию сильно нелинейных процессов, а оценки их параметров можно очень легко получить посредством градиентного спуска. Что справедливо даже для моделей с миллионами параметров. Глубокие нейронные сети особенно привлекательны тем, что можно не волноваться о конкретной функции, используемой для представления данных, — квадратичной, сплайновой или какой-либо еще. Глубокая нейросетевая модель — это и универсальное средство аппроксимации, и удобный метод оценки параметров. Причем это средство аппроксимации можно подстроить под наши нужды в контексте разрешающих возможностей модели и ее способности моделировать сложные взаимосвязи

входных/выходных сигналов благодаря компоновке простых «кирпичиков». На рис. 6.6 приведено несколько примеров.

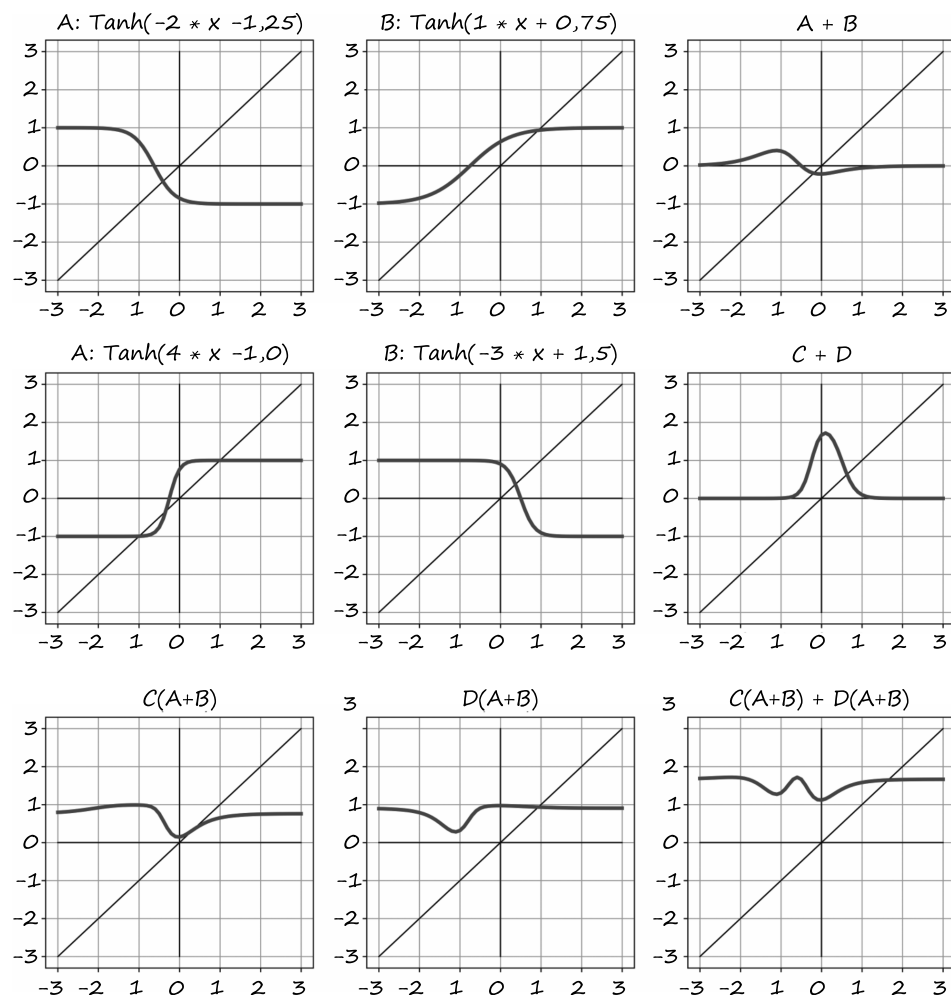


Рис. 6.6. Генерация нелинейных выходных сигналов с помощью различных сочетаний нескольких линейных блоков и функций активации Tanh

На четырех графиках слева сверху показаны четыре нейрона — А, В, С и D, — каждый со своими (произвольно выбранными) весовым коэффициентом и смещением. Все нейроны используют функцию активации Tanh с минимумом -1 и максимумом 1 . Из разных весов и смещений сдвигается центральная точка и меняется степень резкости перехода от минимума к максимуму, но общая форма графика остается прежней. В столбце справа от них показаны результаты

попарного сложения нейронов ($A + B$, а ниже — $C + D$). Здесь уже видны интересные свойства, имитирующие отдельный слой нейронов. $A + B$ демонстрирует слегка напоминающую S кривую, стремящуюся на краях к нулю, но с положительным и отрицательным «горбами» посередине. И наоборот, график $C + D$ содержит лишь большой положительный «горб», с большим значением в максимуме, чем единичный максимум для одного нейрона.

В третьем ряду нейроны сочетаются так, как было бы в двухслойной сети. Как у $C(A + B)$, так и у $D(A + B)$ есть такие же положительный и отрицательный «горбы», как и у $A + B$, но положительный «горб» менее выражен. Сочетание $C(A + B) + D(A + B)$ демонстрирует новое качество: *два* четко выраженных отрицательных «горба» и, похоже, второй, очень слабо выраженный положительный «горб» слева от главной зоны интереса. И для этого достаточно всего лишь четырех нейронов в двух слоях!

Опять же, параметры этих нейронов были выбраны исключительно ради визуальной наглядности. Обучение заключается в поиске приемлемых значений этих весовых коэффициентов и смещений, таких, чтобы итоговая сеть правильно решила поставленную задачу, например предсказание вероятных температур по географическим координатам и времени года. *Успешное выполнение задачи* должно означать получение правильного выходного сигнала для новых данных с помощью того же процесса генерации данных, который использовался для обучения модели. Успешно обученная сеть с помощью значений весов и смещений должна отражать присущую данным структуру в виде осмысленных числовых представлений, дающих правильные результаты для новых данных.

Давайте продвинемся на шаг дальше в понимании механизмов обучения: глубокие нейронные сети позволяют аппроксимировать чрезвычайно нелинейные явления без явной модели. При этом используется универсальная необученная модель, которая затем адаптируется под конкретную задачу с помощью набора входных и выходных сигналов, а также функции потерь для обратного распространения ошибки. Адаптацию обобщенной модели к задаче с помощью примеров данных мы и называем *обучением*, поскольку модель не создавалась под эту конкретную задачу — в нее не закладывалось никаких правил, описывающих эту задачу.

В примере с термометром мы предположили, что оба термометра измеряют температуру линейно. Этим допущением мы неявным образом закодировали правило для нашей задачи: мы жестко задали форму входной/выходной функции, лишив себя возможности аппроксимировать что-либо, кроме точек данных, расположенных вдоль прямой. При росте размерности задачи (количества входных/выходных сигналов) и усложнении взаимосвязей входных/выходных сигналов допущения о форме входной/выходной функции становятся неуместными. Задача физика или специалиста по прикладной математике

зачастую и состоит в том, чтобы придумать функциональное описание явления на теоретической основе, чтобы можно было оценить неизвестные параметры по измеренным значениям и получить точную модель окружающего мира. Глубокие нейронные сети же представляют собой семейства функций, способных аппроксимировать широкий спектр взаимосвязей входных/выходных сигналов, не требуя от нас придумывать поясняющую конкретное явление модель. В каком-то смысле мы отказываемся от пояснения в обмен на возможность решения все более сложных задач. С другой стороны, зачастую нам просто не хватает способностей, информации или вычислительных ресурсов для создания явной модели задачи, так что единственный выход — это методы, ориентированные на работу с данными.

6.2. МОДУЛЬ NN PYTORCH

После всех этих разговоров о нейронных сетях вам, наверное, интересно создать нейронную сеть с нуля на PyTorch. Первое, что нужно сделать, — это заменить нашу линейную модель блоком нейронной сети. В какой-то мере это шаг назад, ведь мы уже убедились, что для разметки нашего термометра достаточно линейной функции, но будет поучительно начать с достаточно простой задачи и затем перейти к более сложным.

В PyTorch есть отдельный подмодуль, посвященный нейронным сетям, — `torch.nn`. Он включает «кирпичики», необходимые для создания всех видов нейросетевых архитектур. В терминологии PyTorch эти «кирпичики» называются *модулями* (в других фреймворках подобные стандартные блоки часто называются *слоями* (*layers*)). Модуль PyTorch — это класс Python, наследующий базовый класс `nn.Module`. Атрибутами модуля PyTorch могут быть один или несколько экземпляров класса `Parameter` — тензоров, значения которых проходят оптимизацию во время процесса обучения (аналогично `w` и `b` в нашей линейной модели). У модуля может быть также один или несколько подмодулей (подклассов `nn.Module`) в качестве атрибутов, и он сможет отслеживать также и их параметры.

ПРИМЕЧАНИЕ

Подмодули должны быть атрибутами верхнего уровня, а не быть «закопаны» внутри экземпляров `list` или `dict`! В противном случае оптимизатор не сможет их найти (а значит, и их параметры). На случай, если модели потребуется список или ассоциативный массив подмодулей, в PyTorch есть классы `nn.ModuleList` и `nn.ModuleDict`.

Неудивительно, что у `nn.Module` есть подкласс `nn.Linear`, применяющий ко входным сигналам аффинное преобразование (через атрибуты-параметры `weight` и `bias`) и эквивалентный реализованному нами выше в экспериментах с термометрами. А сейчас мы продолжим точно с того места, на котором закончили и преобразуем наш предшествующий код так, чтобы использовать `nn`.

6.2.1. Использование метода `__call__` вместо метода `forward`

У всех подклассов `nn.Module` в PyTorch есть метод `__call__`, позволяющий создавать экземпляры `nn.Linear` и вызывать их как функции следующим образом (code/p1ch6/1_neural_networks.ipynb):

```
# In[5]:
import torch.nn as nn

linear_model = nn.Linear(1, 1)  ← Мы обсудим аргументы конструктора в ближайшее время
linear_model(t_un_val)

# Out[5]:
tensor([[0.6018],
        [0.2877]], grad_fn=<AddmmBackward>)
```

Вызов экземпляра `nn.Module` с набором инструментов приводит к вызову метода `forward` с теми же аргументами, который реализует прямой проход вычислений, в то время как `__call__` выполняет другие немаловажные операции до и после вызова `forward`. Так что формально можно вызвать `forward` напрямую, и он вернет тот же результат, что и `__call__`, но делать это из пользовательского кода не рекомендуется:

```
y = model(x)  ← Правильно!
y = model.forward(x)  ← Неправильно! Не делайте так!
```

Ниже приведена реализация `Module.__call__` (мы опустили части, связанные с JIT и упростили кое-что для ясности; `torch/nn/modules/module.py`, строка 483, класс `Module`):

```
def __call__(self, *input, **kwargs):
    for hook in self._forward_pre_hooks.values():
        hook(self, input)

    result = self.forward(*input, **kwargs)

    for hook in self._forward_hooks.values():
        hook_result = hook(self, input, result)
        # ...

    for hook in self._backward_hooks.values():
        # ...

    return result
```

Как видите, тут есть множество точек подключения, которые не вызываются должным образом при вызове напрямую метода `forward`.

6.2.2. Обратно к линейной модели

Возвращаемся к нашей линейной модели. Конструктор `nn.Linear` принимает три аргумента: число входных признаков, число выходных признаков и булево значение, указывающее, включает линейная модель смещение или нет (здесь по умолчанию `True`):

```
# In[5]:
import torch.nn as nn

linear_model = nn.Linear(1, 1)
linear_model(t_un_val)
```

Аргументы — входной размер, выходной размер
и смещение по умолчанию равны True

```
# Out[5]:
tensor([[0.6018],
        [0.2877]], grad_fn=<AddmmBackward>)
```

Количество признаков в нашем случае просто отражает размер входного и выходного тензоров, 1 и 1. Если использовать в качестве входных данных и температуру, и атмосферное давление, например, получилось бы два входных признака и один выходной. Как мы увидим, в более сложных моделях с несколькими промежуточными модулями количество признаков связано с мощностью модели.

Итак, у нас есть экземпляр `nn.Linear` с одним входным и одним выходным признаком. Такая модель требует одного весового коэффициента и одного смещения:

```
# In[6]:
linear_model.weight

# Out[6]:
Parameter containing:
tensor([[ -0.0674]], requires_grad=True)

# In[7]:
linear_model.bias

# Out[7]:
Parameter containing:
tensor([0.7488], requires_grad=True)
```

Вызываем модуль с каким-нибудь входным сигналом:

```
# In[8]:
x = torch.ones(1)
linear_model(x)

# Out[8]:
tensor([0.6814], grad_fn=<AddBackward0>)
```

И хотя PyTorch допускает такой вариант, при этом размерность входного сигнала неправильна. У нашей модели — один входной и один выходной сигнал,

но класс `nn.Module` фреймворка PyTorch и его подклассы предназначены для обработки нескольких примеров данных сразу. Для этого модули ожидают наличия во входном сигнале нулевого измерения с количеством примеров данных в *батче* (*batch*). Мы уже говорили об этом в главе 4, когда учились превращать реальные данные в тензоры.

Группировка входных данных в батчи

Все модули в `nn` ориентированы на генерацию выходных сигналов сразу для *батча* из нескольких входных сигналов. Следовательно, если нам нужно заполнить `nn.Linear` для десяти примеров данных, можно создать входной тензор размером $B \times N_{\text{вх}}$, где B — размер батча, а $N_{\text{вх}}$ — число входных признаков, и пропустить его один раз через модель. Например:

```
# In[9]:
x = torch.ones(10, 1)
linear_model(x)

# Out[9]:
tensor([[0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814],
        [0.6814]], grad_fn=<AddmmBackward>)
```

Давайте посмотрим внимательнее, что происходит на рис. 6.7, где показана аналогичная ситуация с организованными в батч данными изображений. Форма входного сигнала $B \times C \times H \times W$ с размером батча 3 (скажем, изображения собаки, птицы и автомобиля), тремя измерениями каналов (красный, зеленый и синий) и неуказанным количеством пикселей по высоте и ширине. Как видим, выходной сигнал представляет собой тензор размера $B \times N_{\text{вых}}$, где $N_{\text{вых}}$ — количество выходных признаков: в данном случае четыре.

Оптимизация по батчам

Причины для организации данных по батчам многогранны. Одна из них — желание полноценно загрузить вычислениями имеющиеся вычислительные ресурсы. В частности, GPU позволяют сильно распараллеливать вычисления, так что при одиночном входном сигнале для маленькой модели большинство вычислительных элементов будет простаивать. При использовании батчей входных сигналов вычисления распределяются по простаивающим в противном случае элементам, так что результаты для батча возвращаются так же быстро, как и одиночный результат. Еще одно преимущество в том, что некоторые развитые

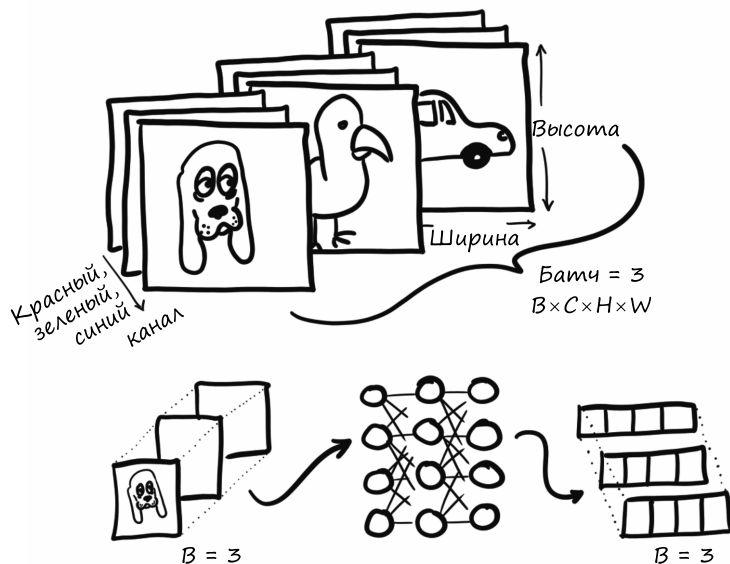


Рис. 6.7. Батч из трех RGB-изображений, подаваемый на вход нейронной сети. Выходной сигнал представляет собой батч из трех векторов размера 4

модели способны использовать статистическую информацию по целому батчу, и эти статистические показатели будут точнее при большом размере батча.

Вернемся к нашим показаниям термометра, где t_u и t_c — два одномерных тензора размером B . Благодаря транслированию мы могли записать нашу линейную модель в виде $w * x + b$, где w и b — два скалярных параметра. И все это благодаря наличию лишь одного входного признака: если бы их было два, нам пришлось бы добавить еще одно измерение, чтобы превратить одномерный тензор в матрицу, с примерами данных по строкам и признакам по столбцам.

Именно это нам и нужно сделать при переходе на использование `nn.Linear`. Меняем форму наших входных сигналов размером B на $B \times N_{ax}$, где $N_{ax} = 1$. Это можно легко сделать при помощи метода `unsqueeze`:

```
# In[2]:
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c).unsqueeze(1)
t_u = torch.tensor(t_u).unsqueeze(1)  # Добавляем еще одно измерение на оси 1

t_u.shape

# Out[2]:
torch.Size([11, 1])
```

Готово! А теперь преобразуем код обучения. Прежде всего, заменим нашу самодельную модель на `nn.Linear(1,1)`, после чего передадим параметры линейной модели оптимизатору:

```
# In[10]:
linear_model = nn.Linear(1, 1)  ← Просто переопределение
optimizer = optim.SGD(
    linear_model.parameters(),  ← Этот вызов метода — вместо [params]
    lr=1e-2)
```

Ранее мы сами должны были создавать параметры и передавать их в качестве первого аргумента методу `optim.SGD`. Теперь можно воспользоваться методом `parameters`, чтобы запросить у любого экземпляра `nn.Module` список параметров его или любого из его подмодулей:

```
# In[11]:
linear_model.parameters()

# Out[11]:
<generator object Module.parameters at 0x7f94b4a8a750>

# In[12]:
list(linear_model.parameters())

# Out[12]:
[Parameter containing:
  tensor([[0.7398]], requires_grad=True), Parameter containing:
  tensor([0.7974], requires_grad=True)]
```

Этот вызов рекурсивно заходит в подмодули, указанные в конструкторе `init` модуля, и возвращает плоский список всех найденных параметров, благодаря чему можно удобно передать его конструктору оптимизатора, как мы и сделали ранее.

Мы можем уже заняться циклом обучения. Оптимизатору передается список тензоров, описанных с `requires_grad = True` — все объекты `Parameter` описываются подобным образом по определению, поскольку их необходимо оптимизировать путем градиентного спуска. При вызове метода `training_loss.backward()` в листьях графа вычислений накапливаются градиенты. Это как раз и есть передаваемые оптимизатору параметры.

Теперь у оптимизатора SGD есть все необходимое. При вызове `optimizer.step()` программа проходит по всем объектам `Parameter` и меняет их на соответствующую содержимому атрибута `grad` долю. Прекрасная архитектура.

Взглянем теперь на сам цикл обучения:

```
# In[13]:
def training_loop(n_epochs, optimizer, model, loss_fn, t_u_train, t_u_val,
                  t_c_train, t_c_val):
    for epoch in range(1, n_epochs + 1):
```

```

t_p_train = model(t_u_train)
loss_train = loss_fn(t_p_train, t_c_train)

t_p_val = model(t_u_val)
loss_val = loss_fn(t_p_val, t_c_val)

optimizer.zero_grad()
loss_train.backward()
optimizer.step()

if epoch == 1 or epoch % 1000 == 0:
    print(f"Epoch {epoch}, Training loss {loss_train.item():.4f},"
          f" Validation loss {loss_val.item():.4f}")

```

Вместо отдельных параметров теперь передается модель

Функция потерь также передается. Мы воспользуемся ею очень скоро

Практически ничего не поменялось, за исключением того, что теперь не нужно передавать `params` явным образом в `model`, поскольку параметры модели содержатся внутри нее самой.

Остался еще один, последний элемент `torch.nn`, которым мы можем воспользоваться: функция потерь. Разумеется, `nn` включает несколько распространенных функций потерь, одна из которых — `nn.MSELoss`, ее мы раньше описали в качестве нашей `loss_fn`. Функции потерь в `nn` — тоже подклассы `nn.Module`, так что мы создадим еще один экземпляр и вызовем его как функцию. В нашем случае мы заменяем самодельную `loss_fn`:

```

# In[15]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(linear_model.parameters(), lr=1e-2)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    model = linear_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print()
print(linear_model.weight)
print(linear_model.bias)

# Out[15]:
Epoch 1, Training loss 134.9599, Validation loss 183.1707
Epoch 1000, Training loss 4.8053, Validation loss 4.7307
Epoch 2000, Training loss 3.0285, Validation loss 3.0889
Epoch 3000, Training loss 2.8569, Validation loss 3.9105

Parameter containing:
tensor([[5.4319]], requires_grad=True)
Parameter containing:
tensor([-17.9693], requires_grad=True)

```

Мы больше не используем нашу рукописную функцию потерь

Все остальное в нашем цикле обучения остается без изменений. Даже получаемые результаты не меняются. Конечно, мы и ожидали таких же результатов, ведь отличия означали бы программную ошибку в одной из двух реализаций.

6.3. НАКОНЕЦ-ТО НЕЙРОННАЯ СЕТЬ

Это был долгий путь — чтобы написать 20 строк кода, необходимых для описания и обучения модели, нам пришлось изучить немало всего. Надеемся, что теперь все белые пятна, связанные с обучением модели, заполнены и осталась одна техника. Изученное позволит нам быть хозяевами своего кода, а не просто стучать по черному ящику в затруднительных ситуациях.

Остался один последний шаг: заменить нашу линейную модель нейронной сетью в качестве аппроксимирующей функции. Мы уже говорили, что нейронная сеть не гарантирует лучшей модели, поскольку процесс, лежащий в основе нашей задачи градуировки, по своей сущности линейный. Однако полезно будет перейти от линейной модели к нейронной сети в контролируемой среде, чтобы потом не запутаться.

6.3.1. Замена линейной модели

Мы не будем менять больше ничего, включая функцию потерь, только поменяем модель. Давайте создадим простейшую возможную нейронную сеть: линейный модуль с последующей функцией активации, выходной сигнал которого служит входным сигналом другого линейного модуля. Первый¹ линейный слой + функция активации обычно называется *скрытым (hidden) слоем* по историческим причинам, поскольку их выходной сигнал не виден пользователю непосредственно, а подается на вход выходного слоя. И хотя входной и выходной сигналы модели — размером 1 (один входной и один выходной признак), размер выходного сигнала первого линейного модуля обычно больше 1. Как вы помните из вышеприведенных пояснений о роли функций активации, в результате этого различные нейроны могут реагировать на различные диапазоны входного сигнала, что повышает разрешающие возможности модели. Последний линейный слой получает на входе выходной сигнал функций активации и возвращает их линейное сочетание в виде выходного сигнала.

Не существует стандартного способа визуального изображения нейронных сетей. На рис. 6.8 показаны два более или менее классических способа: слева показано, как нашу сеть могли бы изобразить в руководстве для начинающих,

¹ В нашем случае вообще в сети, конечно, может быть много скрытых слоев. — *Примеч. пер.*

а стиль, подобный изображенному справа, присущ скорее более продвинутой литературе и научным статьям. Достаточно часто блоки на схеме приближенно соответствуют модулям PyTorch нейронной сети (хотя иногда такие вещи, как слой активации Tanh, явным образом не отображаются). Обратите внимание на одно довольно тонкое различие между этими двумя способами: на графе слева входные сигналы и (промежуточные) результаты отображаются в кружках, как основные элементы. Справа же показаны более укрупненные шаги вычислений.

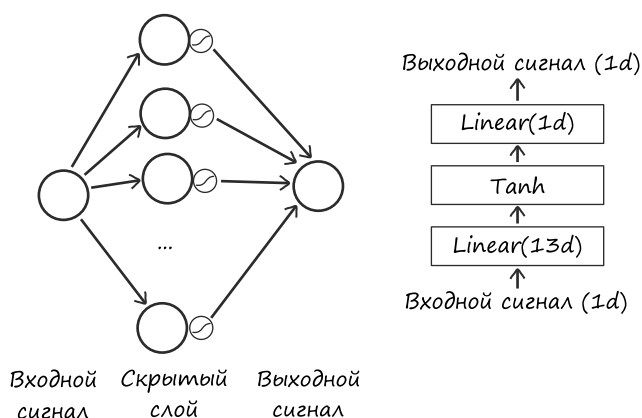


Рис. 6.8. Два представления нашей простейшей нейронной сети.
Слева: версия для начинающих. Справа: более продвинутая версия

Модуль `nn` предоставляет удобный способ соединения модулей цепочкой с помощью контейнера `nn.Sequential`:

```
# In[16]:
seq_model = nn.Sequential(
    nn.Linear(1, 13),
    nn.Tanh(),
    nn.Linear(13, 1))

seq_model
```

Значение 13 взято произвольно.
Мы хотели, чтобы размер отличался
от других размеров имеющихся
у нас тензорных форм

```
# Out[16]:
Sequential(
  (0): Linear(in_features=1, out_features=13, bias=True)
  (1): Tanh()
  (2): Linear(in_features=13, out_features=1, bias=True)
)
```

В итоге получается модель, принимающая входные сигналы, ожидаемые первым указанным в виде аргумента `nn.Sequential` модулем, передает промежуточные выходные сигналы следующему модулю и выдает возвращаемый последним

модулем выходной сигнал. Модель переходит от одного входного признака до 13 скрытых признаков, пропускает их через функцию активации `Tanh` и, наконец, объединяет получившиеся 13 чисел в один выходной признак.

6.3.2. Просматриваем информацию о параметрах

При вызове `model.parameters()` собираются `weight` и `bias` как из первого, так и из второго линейного модуля. В данном случае будет поучительно изучить параметры, отобразив на экран информацию об их форме:

```
# In[17]:
[param.shape for param in seq_model.parameters()]

# Out[17]:
[torch.Size([13, 1]), torch.Size([13]), torch.Size([1, 13]), torch.Size([1])]
```

Эти тензоры и получит оптимизатор. Опять же, после вызова `model.backward()` все параметры заполняются `grad`, а затем оптимизатор обновляет их значения соответствующим образом во время вызова `optimizer.step()`. Не так уж сильно отличается от нашей предыдущей модели, правда? В конце концов, они обе — дифференцируемые модели, обучаемые посредством градиентного спуска.

Несколько замечаний относительно параметров `nn.Module`. При просмотре информации о параметрах модели, состоящей из нескольких модулей, не мешает возможность идентифицировать параметры по названию. Для этой цели предназначен метод `named_parameters`:

```
# In[18]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[18]:
0.weight torch.Size([13, 1])
0.bias torch.Size([13])
2.weight torch.Size([1, 13])
2.bias torch.Size([1])
```

Названия модулей в `Sequential` представляют собой просто порядковые номера модулей в списке аргументов. Что любопытно, `Sequential`¹ также принимает на входе `OrderedDict`, в котором можно указать название каждого из передаваемых `Sequential` модулей:

¹ Не во всех версиях Python определен порядок прохода в цикле по `dict`, так что мы используем тут `OrderedDict`, чтобы гарантировать правильный порядок слоев и подчеркнуть, что порядок слоев важен.

```
# In[19]:
from collections import OrderedDict

seq_model = nn.Sequential(OrderedDict([
    ('hidden_linear', nn.Linear(1, 8)),
    ('hidden_activation', nn.Tanh()),
    ('output_linear', nn.Linear(8, 1))
]))

seq_model

# Out[19]:
Sequential(
  (hidden_linear): Linear(in_features=1, out_features=8, bias=True)
  (hidden_activation): Tanh()
  (output_linear): Linear(in_features=8, out_features=1, bias=True)
)
```

Это дает возможность дать подмодулям более понятные названия:

```
# In[20]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[20]:
hidden_linear.weight torch.Size([8, 1])
hidden_linear.bias torch.Size([8])
output_linear.weight torch.Size([1, 8])
output_linear.bias torch.Size([1])
```

Намного информативнее; но нам еще нужно добиться большей гибкости процесса движения данных по сети, который пока что остается чисто последовательным — название `nn.Sequential` говорит само за себя. В главе 8 мы обсудим, как получить полный контроль над обработкой входных данных путем создания подкласса `nn.Module`.

Обращаться к конкретным объектам `Parameter` можно путем указания подмодулей в качестве атрибутов:

```
# In[21]:
seq_model.output_linear.bias

# Out[21]:
Parameter containing:
tensor([-0.0173], requires_grad=True)
```

Это очень удобно для просмотра параметров или их градиентов, например для мониторинга градиентов во время обучения, как мы делали в начале данной главы. Допустим, мы хотели бы вывести на экран градиенты параметра `weight` линейной части скрытого слоя. Запускаем цикл обучения для новой модели нейронной сети, после чего смотрим на получившиеся градиенты после последней эпохи:

```
# In[22]:
optimizer = optim.SGD(seq_model.parameters(), lr=1e-3)
training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    model = seq_model,
    loss_fn = nn.MSELoss(),
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print('output', seq_model(t_un_val))
print('answer', t_c_val)
print('hidden', seq_model.hidden_linear.weight.grad)

# Out[22]:
Epoch 1, Training loss 182.9724, Validation loss 231.8708
Epoch 1000, Training loss 6.6642, Validation loss 3.7330
Epoch 2000, Training loss 5.1502, Validation loss 0.1406
Epoch 3000, Training loss 2.9653, Validation loss 1.0005
Epoch 4000, Training loss 2.2839, Validation loss 1.6580
Epoch 5000, Training loss 2.1141, Validation loss 2.0215
output tensor([[ -1.9930],
               [20.8729]], grad_fn=<AddmmBackward>)
answer tensor([[ -4.],
               [21.]])
hidden tensor([[ 0.0272],
               [ 0.0139],
               [ 0.1692],
               [ 0.1735],
               [-0.1697],
               [ 0.1455],
               [-0.0136],
               [-0.0554]])
```

Мы немного снизили скорость обучения, чтобы повысить стабильность

6.3.3. Сравнение с линейной моделью

Запустим модель на всем массиве данных и посмотрим, насколько она отклоняется от прямой:

```
# In[23]:
from matplotlib import pyplot as plt

t_range = torch.arange(20., 90.).unsqueeze(1)

fig = plt.figure(dpi=600)
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
plt.plot(t_range.numpy(), seq_model(0.1 * t_range).detach().numpy(), 'c-')
plt.plot(t_u.numpy(), seq_model(0.1 * t_u).detach().numpy(), 'kx')
```

Результат приведен на рис. 6.9. Принимаем во внимание, что нейронная сеть склонна к переобучению, как мы обсуждали в главе 5, поскольку стремится приблизиться к измерениям, в том числе и к зашумленным. Даже у нашей крошечной нейронной сети слишком много параметров для подгонки к небольшому числу имеющихся у нас показаний термометра. В целом, впрочем, она работает не так уж плохо.

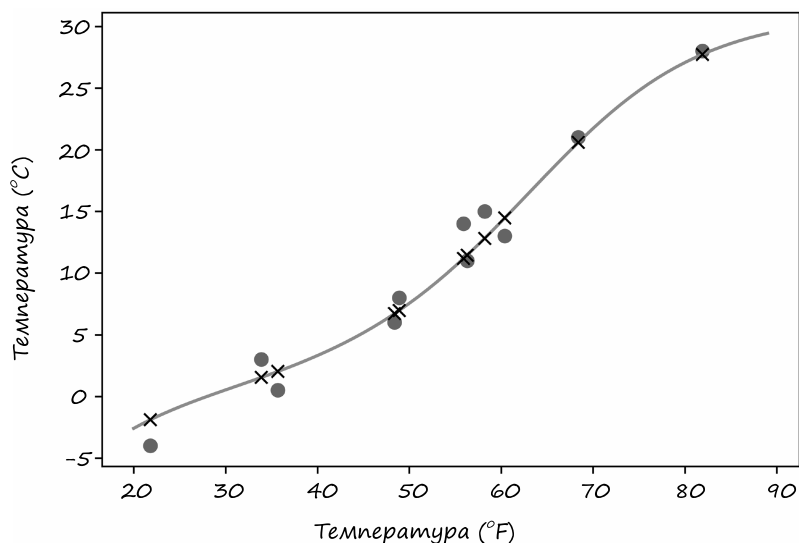


Рис. 6.9. График нейросетевой модели со входными данными (кружки) и выходным сигналом модели (X). Непрерывная кривая отражает поведение модели в промежутках между примерами данных

6.4. ИТОГИ ГЛАВЫ

Мы охватили немало материала в главах 5 и 6, хотя обсуждали очень простую задачу. Мы подробно разобрали вопросы создания дифференцируемых моделей и их обучения с использованием градиентного спуска, сначала с помощью одного `autograd`, а затем на основе модуля `nn`. Теперь вы уже должны хорошо себе представлять, что происходит «за кулисами» обучения. Надеемся, вы достаточно распробовали `PyTorch`, чтобы захотеть еще!

6.5. УПРАЖНЕНИЯ

1. Поэкспериментируйте с количеством скрытых нейронов в нашей простой нейросетевой модели, а также со скоростью обучения.

- А. Какие изменения приводят к повышению линейности выходного сигнала модели?
 - Б. Можете ли вы добиться явного переобучения модели?
2. Третья по сложности задача физики — выбор подходящего вина для празднования открытий. Загрузите данные о винах из главы 4 и создайте новую модель с соответствующим количеством входных параметров.
- А. Сколько времени занимает обучение по сравнению с задачей о термометре?
 - Б. Можете ли вы объяснить, какие факторы влияют на длительность обучения?
 - В. Можете ли вы добиться снижения потерь при обучении на этом наборе данных?
 - Г. Как построить график для этого набора данных?

6.6. РЕЗЮМЕ

- Нейронные сети могут автоматически адаптироваться к поставленной задаче.
- Нейронные сети обеспечивают возможность удобного доступа к аналитически выраженным производным функции потерь относительно любого из параметров модели, что позволяет очень эффективно обновлять эти параметры. Благодаря механизму автоматического дифференцирования PyTorch с легкостью предоставляет подобные производные.
- Функции активации, в дополнение к линейным преобразованиям, позволяют нейронным сетям аппроксимировать сильно нелинейные функции, оставляя их при этом достаточно простыми для оптимизации.
- Модуль `nn`, вместе со стандартной библиотекой для работы с тензорами, обеспечивает все необходимое для создания нейронных сетей.
- Чтобы выявить переобучение, необходимо отделять обучающий набор точек данных от проверочного набора. Не существует единого рецепта борьбы с переобучением, но для начала имеет смысл найти дополнительные или более разнообразные данные, а также ограничиться более простыми моделями.
- Всем, кто занимается исследованием данных, следует постоянно визуализировать их.

Различаем птиц и самолеты: обучение на изображениях

В этой главе

- ✓ Создаем сеть прямого распространения.
- ✓ Загрузка данных с помощью объектов Dataset и DataLoader.
- ✓ Функции потерь для классификации.

В предыдущей главе мы смогли заглянуть внутрь механизмов обучения с помощью градиентного спуска, а также предоставляемых PyTorch возможностей по созданию и оптимизации моделей. Для демонстрации мы использовали простую модель регрессии с одним входным сигналом и одним выходным, позволившую нам обеспечить ясность изложения, но, признаем, далеко не такую уж интересную.

В этой главе мы продолжим формировать основы нашего понимания нейронных сетей. На этот раз мы обратим внимание на изображения. Именно благодаря задачам распознавания изображений весь мир осознал потенциал глубокого обучения.

Мы будем подступать к решению простой задачи распознавания изображений шаг за шагом на основе простой нейронной сети, подобной той, которую мы видели в предыдущей главе. На этот раз вместо крошечного набора числовых данных мы воспользуемся более обширным набором крошечных изображений. Давайте сначала скачаем этот набор данных и подготовим его к использованию.

7.1. НАБОР КРОШЕЧНЫХ ИЗОБРАЖЕНИЙ

Нет ничего лучше интуитивного понимания предмета и нет лучше способа достичь такого понимания, чем работа с простыми данными. Один из простейших наборов данных для распознавания изображений — набор изображений рукописных цифр, известный под названием MNIST. Здесь же мы воспользуемся другим набором данных, относительно простым, но намного более интересным. Он называется CIFAR-10 и, подобно схожему с ним CIFAR-100, почти десятилетие считается классическим в сфере машинного зрения.

CIFAR-10 состоит из 60 000 крошечных (32×32) цветных (RGB) изображений, маркированных цифрами, соответствующими одному из десяти классов: самолет (0), автомобиль (1), птица (2), кошка (3), олень (4), собака (5), лягушка (6), лошадь (7), корабль (8) и грузовик (9)¹. Сейчас CIFAR-10 считается слишком простым для разработки или проверки новых научных исследований, но для наших учебных целей подходит прекрасно. Мы воспользуемся модулем `torchvision` для автоматического скачивания этого набора данных и загрузки его в виде набора тензоров PyTorch. На рис. 7.1 показан кусочек CIFAR-10.



Рис. 7.1. Примеры изображений из всех классов CIFAR-10

7.1.1. Скачиваем CIFAR-10

Как мы и намеревались сделать, давайте импортируем `torchvision` и воспользуемся модулем `datasets` для скачивания данных CIFAR-10:

¹ Эти изображения были собраны и маркированы Крижевским (Krizhevsky), Наиром (Nair) и Хинтоном (Hinton) из Канадского института перспективных исследований (Canadian Institute For Advanced Research, CIFAR) из более крупного набора немаркированных цветных изображений размером 32×32 : «Набор из 80 миллионов крошечных изображений» от Лаборатории вычислительной техники и искусственного интеллекта (Computer Science and Artificial Intelligence Laboratory, CSAIL) Массачусетского технологического института.

Создает объект `Dataset` для обучающих данных;
если данные отсутствуют — `TorchVision` скачивает их

```
# In[2]:
from torchvision import datasets
data_path = '../data-unversioned/p1ch7/'
→ cifar10 = datasets.CIFAR10(data_path, train=True, download=True)
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True) ←
```

При `train=False` получаем проверочный
набор данных, при необходимости снова
производится скачивание

Первый из передаваемых в функцию `CIFAR10` аргументов — это место, откуда нужно скачать данные¹; второй указывает, необходим нам обучающий набор данных или проверочный; а третий — разрешаем ли мы PyTorch скачать данные, если они отсутствуют в указанном в первом аргументе месте.

Аналогично `CIFAR10` подмодуль `datasets` предоставляет доступ и к другим наиболее популярным наборам данных машинного зрения, в частности MNIST, Fashion-MNIST, CIFAR-100, SVHN, Coco и Omniglot. Во всех случаях набор данных возвращается в виде подкласса `torch.utils.data.Dataset`. Как можно видеть, он указан в качестве базового класса в порядке разрешения методов нашего экземпляра `cifar10`:

```
# In[4]:
type(cifar10).__mro__

# Out[4]:
(torchvision.datasets.cifar.CIFAR10,
torchvision.datasets.vision.VisionDataset,
torch.utils.data.dataset.Dataset,
object)
```

7.1.2. Класс `Dataset`

Пора выяснить, что на практике значит «подкласс `torch.utils.data.Dataset`». Из рис. 7.2 понятно, что представляет собой объект `Dataset` PyTorch. Он должен реализовывать два метода: `__len__` и `__getitem__`, первый из которых должен возвращать количество элементов набора данных, а второй — элемент данных, состоящий из примера данных и соответствующей метки (числового индекса)².

На практике, когда у объекта Python есть метод `__len__`, его можно передать в качестве аргумента встроенной функции `len` языка Python:

¹ На самом деле в этом аргументе указывается место, где находится набор данных или куда он будет скачиваться в случае отсутствия: <https://pytorch.org/vision/0.11/datasets.html#cifar>. — *Примеч. пер.*

² Для более опытных пользователей в PyTorch есть также класс `IterableDataset` для наборов, где произвольный доступ к данным требует слишком много ресурсов или не имеет смысла: например, когда данные генерируются динамически, во время работы.

```
# In[5]:
len(cifar10)
```

```
# Out[5]:
50000
```

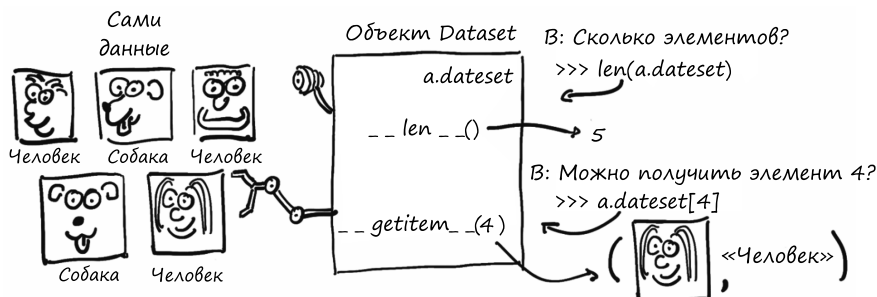


Рис. 7.2. Основная идея объекта Dataset PyTorch: он не обязательно содержит данные, но предоставляет к ним доступ через методы `__len__` и `__getitem__`

Аналогично, поскольку у объекта Dataset есть метод `__getitem__`, можно воспользоваться стандартным способом доступа по индексу к кортежам и спискам для обращения к отдельным элементам. В данном случае мы получаем изображение типа PIL (библиотека Python для работы с изображениями, пакет PIL), а также интересующий нас результат — целочисленное значение 1, соответствующее метке «автомобиль»¹:

```
# In[6]:
img, label = cifar10[99]
img, label, class_names[label]

# Out[6]:
(<PIL.Image.Image image mode=RGB size=32x32 at 0x7FB383657390>,
1,
'automobile')
```

Итак, пример данных из набора `data.CIFAR10` представляет собой экземпляр изображения PIL RGB. Можно сразу же вывести его на экран²:

```
# In[7]:
plt.imshow(img)
plt.show()
```

¹ Авторы пропустили в коде описание переменной `class_names`:
`class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
'dog', 'frog', 'horse', 'ship', 'truck']`

Примеч. пер.

² Не забудьте для этого импортировать библиотеку `pyplot`:
`from matplotlib import pyplot as plt`

Примеч. пер.

Получаем результат, приведенный на рис. 7.3. Красная машина!¹

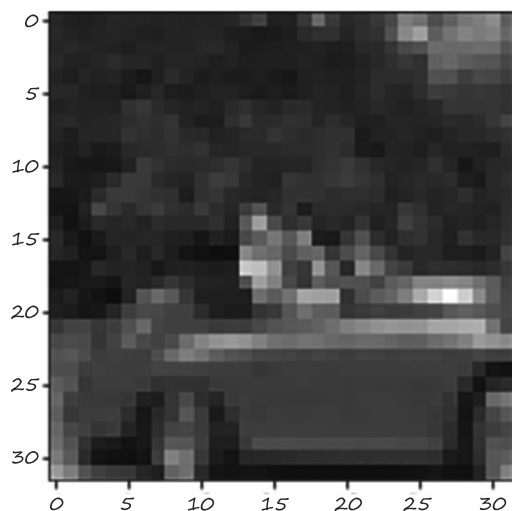


Рис. 7.3. Девяносто девятое изображение из набора данных CIFAR-10: автомобиль

7.1.3. Преобразования объектов Dataset

Все это хорошо, но нам нужно преобразовать изображение PIL в тензор PyTorch, чтобы с ним что-то делать. Тут нам пригодится модуль `torchvision.transforms`, где описан набор компонентных и подобных функций объектов, которые можно передавать в качестве аргументов в наборы данных `torchvision`, например `datasets.CIFAR10(...)`, и преобразовывать данные после их загрузки, но перед возвратом их методом `__getitem__`. Вот список доступных объектов:

```
# In[8]:
from torchvision import transforms
dir(transforms)
```

```
# Out[8]:
['CenterCrop',
 'ColorJitter',
 ...
 'Normalize',
 'Pad',
 'RandomAffine',
 ...
 'RandomResizedCrop',
 'RandomRotation',
 'RandomSizedCrop',
 ...]
```

¹ В напечатанном виде выглядит не так впечатляюще; вам придется поверить нам на слово либо заглянуть в блокнот Jupyter.

```
'TenCrop',
'ToPILImage',
'ToTensor',
...
]
```

Среди этих преобразований можно заметить `ToTensor`, превращающее массивы NumPy и изображения PIL в тензоры. Оно также располагает измерения выходного тензора в порядке $C \times H \times W$ (каналы, высота, ширина; точно как у нас в главе 4).

Попробуем преобразование `ToTensor` в действии. После создания экземпляра его можно вызвать как обычную функцию, передав изображение PIL в качестве аргумента, и получить в качестве результата тензор:

```
# In[9]:
from torchvision import transforms

to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
img_t.shape

# Out[9]:
torch.Size([3, 32, 32])
```

Изображение превратилось в тензор формы $3 \times 32 \times 32$, то есть в изображение размером 32×32 с тремя цветовыми каналами (RGB).

Как мы и предполагали, можно передать это преобразование непосредственно в виде аргумента `dataset.CIFAR10`:

```
# In[10]:
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
                                  transform=transforms.ToTensor())
```

Теперь при обращении к элементу набора данных будет возвращаться тензор, а не изображение PIL:

```
# In[11]:
img_t, _ = tensor_cifar10[99]
type(img_t)

# Out[11]:
torch.Tensor
```

Как и можно было ожидать, первое измерение этого тензора является каналом, а тип скалярных значений — `float32`:

```
# In[12]:
img_t.shape, img_t.dtype

# Out[12]:
(torch.Size([3, 32, 32]), torch.float32)
```

Если значения в исходном изображении PIL находились в диапазоне от 0 до 255 (8 бит на канал), то ToTensor преобразует данные в 32-битные значения с плавающей запятой на канал, масштабируя их к диапазону от 0,0 до 1,0. Проверим это:

```
# In[13]:
img_t.min(), img_t.max()

# Out[13]:
(tensor(0.), tensor(1.))
```

Проверим также, что на выходе получается то же самое изображение:

```
# In[14]:
plt.imshow(img_t.permute(1, 2, 0))  ← Порядок осей координат меняется с C×H×W на H×W×C
plt.show()

# Out[14]:
<Figure size 432x288 with 1 Axes>
```

Как видим на рис. 7.4, получается то же изображение, что и раньше.

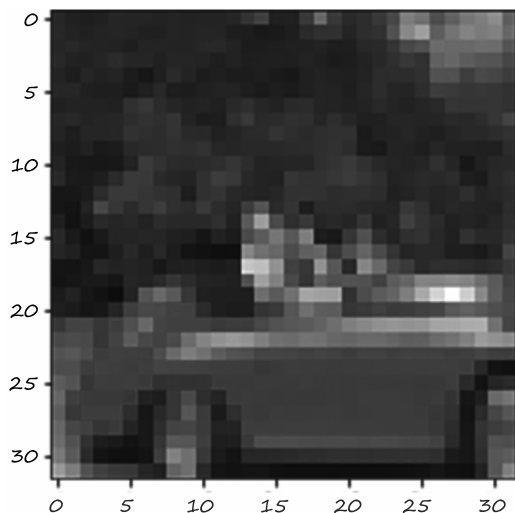


Рис. 7.4. Мы уже видели это раньше

Все правильно. Обратите внимание, что нам пришлось воспользоваться `permute`, чтобы поменять порядок осей координат $C \times H \times W$ на $H \times W \times C$, ожидаемый Matplotlib.

7.1.4. Нормализация данных

Преобразования очень удобны, поскольку их можно организовывать цепочкой с помощью `transforms.Compose` и они обеспечивают четкую и ясную нормализацию

и дополнение данных непосредственно при их загрузке. Например, рекомендуемой практикой считается нормализовать набор данных до нулевого среднего значения и единичного стандартного отклонения по каждому из каналов. Мы уже упоминали это в главе 4, но теперь, после главы 5, понимаем почему: при выборе функций активации, линейных около нуля (плюс-минус 1 или 2), ограничение данных тем же диапазоном повышает вероятность ненулевых градиентов нейронов, а значит, и ускоряет обучение. Кроме того, нормализация каналов к одинаковому распределению гарантирует смешение и обновление информации из разных каналов (посредством градиентного спуска) с одинаковой скоростью обучения. Точно как в подразделе 5.4.4, когда мы приводили весовой коэффициент к тому же порядку величины, что и смещение, в нашей модели преобразования температур.

Чтобы обеспечить нулевое среднее значение и единичное стандартное отклонение по каждому из каналов, необходимо вычислить среднее значение и стандартное отклонение каждого из каналов набора данных и применить следующее преобразование: $v_n[c] = (v[c] - \text{mean}[c]) / \text{stdev}[c]$. Именно это и делает преобразование `transforms.Normalize`. Значения `mean` и `stdev` необходимо вычислить отдельно (преобразование их не вычисляет). Давайте вычислим их для обучающего набора данных CIFAR-10.

Поскольку набор данных CIFAR-10 невелик, можно работать с ним полностью в оперативной памяти. Разместим все возвращаемые объектом `Dataset` тензоры последовательно в дополнительном измерении:

```
# In[15]:
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)
imgs.shape

# Out[15]:
torch.Size([3, 32, 32, 50000])
```

Теперь можно легко вычислить поканальные средние значения:

```
# In[16]:
imgs.view(3, -1).mean(dim=1)

# Out[16]:
tensor([0.4915, 0.4823, 0.4468])
```

Напоминаем, что `view(3, -1)` сохраняет наши три канала, схлопывая все остальные измерения в одно и подбирая подходящий размер. Таким образом, наше изображение размером $3 \times 32 \times 32$ преобразуется в тензор 3×1024 , после чего вычисляется среднее значение каждого из каналов

Стандартное отклонение вычисляется аналогично:

```
# In[17]:
imgs.view(3, -1).std(dim=1)

# Out[17]:
tensor([0.2470, 0.2435, 0.2616])
```

Вычислив эти значения, мы можем произвести преобразование `Normalize`:

```
# In[18]:
transforms.Normalize((0.4915, 0.4823, 0.4468), (0.2470, 0.2435, 0.2616))
```

```
# Out[18]:
Normalize(mean=(0.4915, 0.4823, 0.4468), std=(0.247, 0.2435, 0.2616))
```

добавив его после преобразования ToTensor:

```
# In[19]:
transformed_cifar10 = datasets.CIFAR10(
    data_path, train=True, download=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                              (0.2470, 0.2435, 0.2616))
    ]))
```

Обратите внимание, что, если вывести в этот момент взятое из объекта `Dataset` изображение, мы получим нечто далекое от настоящего изображения:

```
# In[21]:
img_t, _ = transformed_cifar10[99]
```

```
plt.imshow(img_t.permute(1, 2, 0))
plt.show()
```

На рис. 7.5 показана наша нормализованная красная машина. Такая картина возникает потому, что нормализация сдвинула уровни RGB за пределы диапазона $[0, 0, 1, 0]$ и изменила порядки значений каналов. Данные никуда не пропали; просто Matplotlib визуализирует их в черном цвете. Запомним это на будущее.

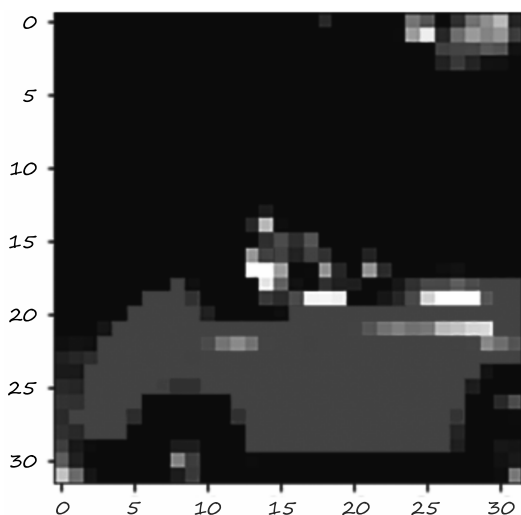


Рис. 7.5. Случайное изображение из CIFAR-10 после нормализации

Итак, мы загрузили интересный набор данных с десятками тысяч изображений! Очень удобно, поскольку нам как раз что-то подобное и понадобится.

7.2. РАЗЛИЧАЕМ ПТИЦ И САМОЛЕТЫ

Джейн, наш друг из орнитологического клуба, установила в лесу к югу от аэропорта множество камер, которые автоматически делают снимок, когда что-то попадает в кадр, и загружают в работающий в реальном времени блог наблюдений за птицами. Проблема в том, что камеры срабатывают на множество самолетов, прилетающих в аэропорт и вылетающих из него. Поэтому Джейн тратит немало времени на удаление из блога фотографий самолетов. Ей необходима автоматизированная система вроде той, что показана на рис. 7.6. Вместо того чтобы удалять фотографии вручную, она хотела бы воспользоваться нейронной сетью — искусственным интеллектом, говоря языком маркетологов, — чтобы сразу же отфильтровывать самолеты.

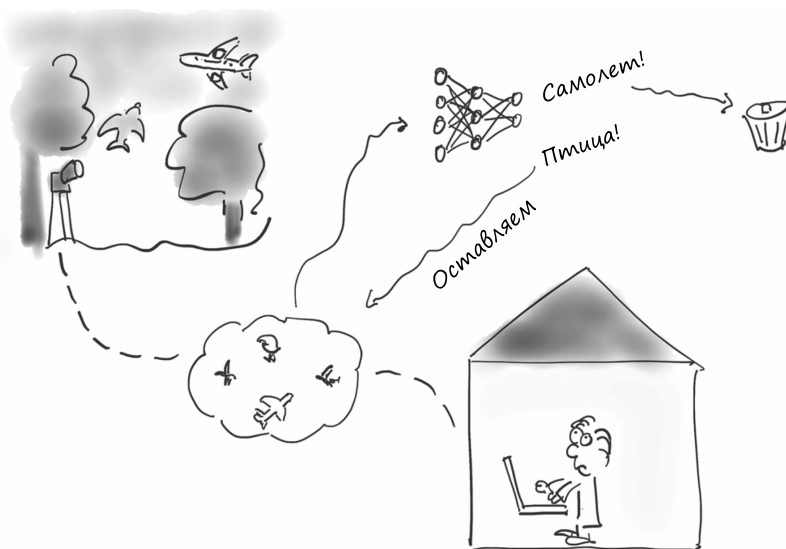


Рис. 7.6. Задача: помочь нашему другу различать птиц и самолеты для ее блога с помощью обучения нейронной сети

Легко! Будет сделано без проблем — у нас как раз есть идеальный набор данных для этой цели (какое совпадение, правда?). Мы выберем всех птиц и самолеты из нашего набора данных CIFAR-10 и создадим нейронную сеть, способную их различать между собой.

7.2.1. Формирование набора данных

Первый этап — приведение данных в правильную форму. Мы создадим подкласс `Dataset`, включающий только птиц и самолеты. Однако набор данных мал, и для работы с ним нам нужен лишь доступ по индексу и `len`. Так что вовсе не обязательно делать подкласс `torch.utils.data.dataset.Dataset`! Почему бы не упростить себе задачу и просто фильтровать данные из `cifar10`¹, перераспределив метки так, чтобы они остались непрерывными? Вот так:

```
# In[5]:
label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
cifar2 = [(img, label_map[label])
           for img, label in cifar10
           if label in [0, 2]]
cifar2_val = [(img, label_map[label])
               for img, label in cifar10_val
               if label in [0, 2]]
```

Объект `cifar2` удовлетворяет основным требованиям к `Dataset` — в нем описаны методы `__len__` и `__getitem__`, — так что мы можем им воспользоваться. Впрочем, осторожнее, мы тут «срезаем углы» с соответствующими ограничениями, так что, возможно, в некоторых случаях имеет смысл реализовать полноценный `Dataset`².

У нас есть набор данных! Далее нам понадобится модель, в которую мы эти данные будем подавать.

7.2.2. Полносвязная модель

Мы научились создавать нейронные сети в главе 5. И знаем, что на входе и выходе ее должны быть тензоры признаков. В конце концов, изображение представляет собой всего лишь набор чисел, расположенных в соответствии с пространственными координатами. Хорошо, мы пока что не знаем, что делать с этими координатами, но чисто теоретически, если вытянуть пиксели изображения в один длинный одномерный вектор, их же можно рассматривать как входные признаки, правда? Этот процесс иллюстрирует рис. 7.7.

¹ Авторы забыли упомянуть, что здесь используется на самом деле уже преобразованный `cifar10`, иначе дальнейший код работать не будет. В прилагаемом блокноте Jupyter все правильно. — *Примеч. пер.*

² Здесь мы создали новый набор данных вручную и переопределили классы. В некоторых случаях достаточно использовать подмножество индексов исходного набора данных, что можно реализовать с помощью класса `torch.utils.data.Subset`. Аналогично существует класс `ConcatDataset`, предназначенный для объединения объектов `Dataset` (состоящих из совместимых элементов) в один больший. В случае итерируемых объектов `Dataset` получить больший и тоже итерируемый `Dataset` можно с помощью `ChainDataset`.

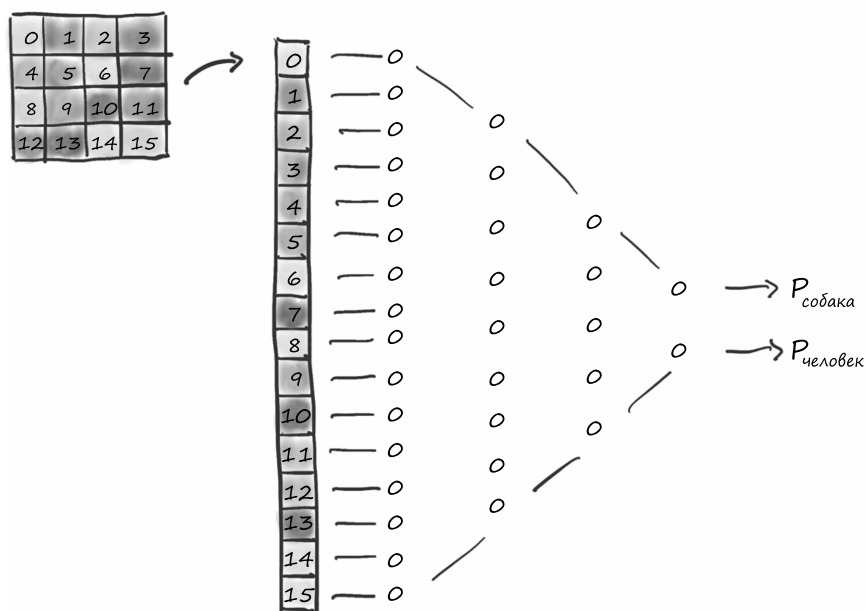


Рис. 7.7. Рассматриваем изображение как одномерный вектор значений, на котором обучаем полносвязный классификатор

Давайте попробуем. Сколько признаков содержит каждый пример данных? Так, $3 \times 32 \times 32$ равняется 3072 входных признака на каждый пример. Начиная с созданной в главе 5 модели, получаем новую модель `nn.Linear` с 3072 входными признаками и некоторым количеством скрытых признаков, за которым следует функция активации, а затем еще один `nn.Linear`, сокращающий модель до соответствующего количества выходных признаков (в данном случае 2):

```
# In[6]:
import torch.nn as nn

n_out = 2

model = nn.Sequential(
    nn.Linear(
        Входные признаки → 3072,
        512,
        ← Размер скрытого слоя
    ),
    nn.Tanh(),
    nn.Linear(
        512,
        ←
        Выходные признаки → n_out,
    )
)
```

Количество 512 скрытых признаков мы выбрали несколько произвольным образом. Нейронной сети требуется по крайней мере один скрытый слой (активации, поэтому два модуля) с нелинейностью между слоями, чтобы сеть могла усваивать произвольные функции так, как мы обсуждали в разделе 6.3, в противном случае модель будет просто линейной. Скрытые признаки отражают (усвоенные сетью) взаимосвязи между входными сигналами, закодированные в матрице весов. Сама по себе модель может научиться «сравнивать» элементы 176 и 208 вектора, но априори она не знает, что необходимо обратить на них внимание, поскольку структурно не знает, что они располагаются в (строка 5, пиксель 16) и (строка 6, пиксель 16), а значит — смежные.

Итак, у нас есть модель. Далее мы обсудим, какими должны быть выходные сигналы нашей модели.

7.2.3. Выходной сигнал классификатора

В главе 6 сеть выдавала в качестве выходного сигнала предсказанную температуру (число с количественным значением). Можно и тут сделать нечто подобное: сделать так, чтобы наша сеть выдавала на выходе одно скалярное значение (то есть `n_out = 1`), привести тип меток к `float` (0,0 для самолета и 1,0 для птицы) и использовать их в качестве целевой величины для `MSELoss` (среднее значение квадратов разностей в батче). При этом задача сведется к задаче регрессии. Однако если взглянуть внимательнее, станет ясно, что мы имеем дело с чем-то принципиально другим¹.

Необходимо понять, что выходной сигнал носит категориальный характер: птица или самолет (либо что-то еще, если мы используем все десять исходных классов). Как мы узнали из главы 4, для представления категориальной величины следует воспользоваться унитарным ее кодированием, например, `[1, 0]` для самолета и `[0, 1]` для птицы (порядок выбран произвольно). Такая схема подходит и в случае десяти классов, как в полном наборе данных CIFAR-10; просто вектор будет длиной 10².

¹ Вместо того чтобы использовать `MSELoss` для номеров классов — что, как вы помните из обсуждения типов значений во врезке «Непрерывные, порядковые и категориальные значения» в главе 4, бессмысленно в случае категорий и на практике вообще не работает, — намного лучше вычислять расстояние между «вероятностными» векторами. Опять же, `MSELoss` очень плохо подходит для задач классификации.

² В нашем частном случае бинарной классификации два значения — это избыточно, поскольку одно всегда равно 1 минус второе. И действительно, PyTorch позволяет выдавать на выходе одно значение вероятности, получая вероятность путем использования в конце модели функции активации `nn.Sigmoid` и функции потерь на основе бинарной перекрестной энтропии `nn.BCELoss`. Существует также `nn.BCELossWithLogits`, объединяющая эти два шага.

В идеальном случае сеть должна выдавать на выходе `torch.tensor([1.0, 0.0])` для самолета и `torch.tensor([0.0, 1.0])` — для птицы. На практике же, поскольку наш классификатор не будет идеален, следует ожидать от сети неких промежуточных значений. Главное в этом случае, что мы можем интерпретировать выходные сигналы как вероятности: первая запись — вероятность класса 'airplane', а вторая — 'bird'.

Изложение задачи на языке вероятностей накладывает несколько дополнительных ограничений на выходные сигналы нашей сети.

- Все элементы выходного сигнала должны находиться в диапазоне $[0.0, 1.0]$ (вероятность исхода не может быть меньше 0 или больше 1).
- Сумма элементов выходного сигнала должна равняться 1.0 (мы уверены, что имеет место один из двух исходов).

Выглядит довольно жестким ограничением на числовой вектор с учетом требования дифференцируемости. Но существует очень ловкий прием, позволяющий его реализовать: *многомерная логистическая функция (softmax)*.

7.2.4. Представление выходного сигнала в качестве вероятностей

Многомерная логистическая функция принимает на входе вектор значений и возвращает другой вектор той же размерности, в котором значения удовлетворяют только что перечисленным ограничениям для вероятностей. Выражение для многомерной логистической функции приведено на рис. 7.8.

Другими словами, мы берем элементы вектора, вычисляем поэлементно экспоненту от них и делим каждый из элементов на сумму экспонент. В виде кода это выглядит примерно следующим образом:

```
# In[7]:
def softmax(x):
    return torch.exp(x) / torch.exp(x).sum()
```

Проверим на входном векторе:

```
# In[8]:
x = torch.tensor([1.0, 2.0, 3.0])

softmax(x)

# Out[8]:
tensor([0.0900, 0.2447, 0.6652])
```

$$0 \leq \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \leq 1$$

Все элементы — в диапазоне от 0 до 1

$$\frac{e^{x_1}}{e^{x_1} + e^{x_2}} + \frac{e^{x_2}}{e^{x_1} + e^{x_2}} = \frac{e^{x_1} + e^{x_2}}{e^{x_1} + e^{x_2}} = 1$$

Сумма элементов равна 1

$$\text{softmax}(x_1, x_2) = \left(\frac{e^{x_1}}{e^{x_1} + e^{x_2}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2}} \right)$$

$$\text{softmax}(x_1, x_2, x_3) = \left(\frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_3}}{e^{x_1} + e^{x_2} + e^{x_3}} \right)$$

⋮

$$\text{softmax}(x_1, \dots, x_n) = \left(\frac{e^{x_1}}{e^{x_1} + \dots + e^{x_n}}, \dots, \frac{e^{x_n}}{e^{x_1} + \dots + e^{x_n}} \right)$$

Рис. 7.8. Многомерная логистическая функция (написано от руки)

Как и ожидалось, полученное удовлетворяет ограничениям, накладываемым на вероятность:

```
# In[9]:
softmax(x).sum()
```

```
# Out[9]:
tensor(1.)
```

Многомерная логистическая функция монотонна, в том смысле, что меньшие входные значения соответствуют меньшим выходным. Однако она не является *масштабно инвариантной* (*scale invariant*), в том смысле, что соотношения значений не сохраняются. На самом деле отношение первого и второго элементов входного сигнала равно 0,5, а отношение тех же элементов выходного сигнала — 0,3678. Это не проблема, поскольку процесс обучения приводит параметры модели к нужным соотношениям.

Многомерная логистическая функция доступна в nn в виде модуля. Поскольку, как обычно, у входных тензоров может быть дополнительное нулевое измерение батчей либо могут быть измерения, по которым кодируются вероятности, а также другие измерения, по которым не кодируются, nn.Softmax требует указания измерения, по которому применяется многомерная логистическая функция:

232 Часть I. Основы PyTorch

```
# In[10]:
softmax = nn.Softmax(dim=1)

x = torch.tensor([[1.0, 2.0, 3.0],
                  [1.0, 2.0, 3.0]])

softmax(x)

# Out[10]:
tensor([[0.0900, 0.2447, 0.6652],
        [0.0900, 0.2447, 0.6652]])
```

В данном случае у нас два входных вектора в двух строках (так же, как когда мы работали по батчам), так что мы указываем измерение 1 в качестве рабочего для `nn.Softmax`.

Замечательно! Можно теперь добавить многомерную логистическую функцию в конец нашей модели, и сеть будет готова выдавать вероятности:

```
# In[11]:
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.Softmax(dim=1))
```

Можно даже попробовать модель в работе, прежде чем ее обучать. Давайте попробуем, просто чтобы посмотреть, что получится. Сначала сформируем батч из одного изображения, птицы (рис. 7.9).

```
# In[12]:
img, _ = cifar2[0]

plt.imshow(img.permute(1, 2, 0))
plt.show()
```

Ой, чуть не забыли. Для вызова модели у входного сигнала должны быть правильные измерения. Вспоминаем, что модель ожидает 3072 входных признака и что `nn` имеет дело с данными, организованными в батчи по нулевому измерению. Так что нам нужно преобразовать наше изображение формы $3 \times 32 \times 32$ в одномерный тензор, а затем добавить еще одно измерение на нулевой позиции. Мы уже знаем, как это делать, из главы 3:

```
# In[13]:
img_batch = img.view(-1).unsqueeze(0)
```

Теперь можно вызывать модель:

```
# In[14]:
out = model(img_batch)
out

# Out[14]:
tensor([[0.4784, 0.5216]], grad_fn=<SoftmaxBackward>)
```

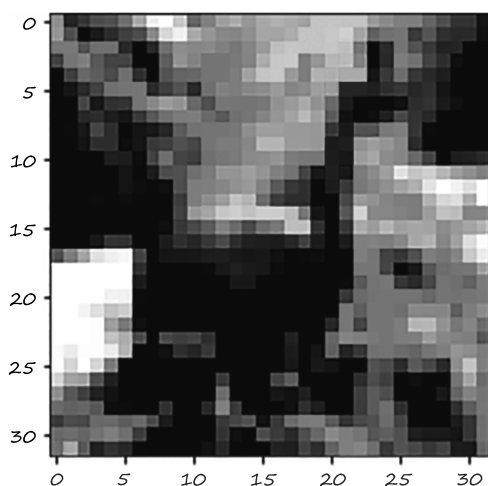


Рис. 7.9. Случайно выбранная птица из набора данных CIFAR-10 (после нормализации)

О, мы получили вероятности! Впрочем, радоваться рано: веса и смещения наших линейных слоев вообще не были обучены. PyTorch взял начальные значения их элементов случайным образом из промежутка от $-1,0$ до $1,0$. Что интересно, в качестве результатов мы получили также `grad_fn` — верхушку графа вычислений обратного прохода (он используется при обратном распространении ошибки)¹.

Кроме того, хотя мы знаем, какая выходная вероятность соответствует какому классу (вспомните наши `class_names`), сеть никак это не демонстрирует. Первая — *airplane*, а вторая — *bird* или наоборот? На данном этапе сеть не знает даже этого. Смысл этим двум числам придает функция потерь после обратного распространения ошибки. Если индексы меток 0 для *airplane* и 1 — для *bird*, то именно такой порядок выходных сигналов и подразумевается. Следовательно, после обучения мы сможем получить метки в виде индексов путем вычисления *аргумента максимизации* (*argmax*) индекса, при котором вероятность достигает максимального значения. Функция `torch.max`, что очень удобно при передаче ей в качестве аргумента измерения, возвращает максимальный элемент по этому измерению, а также соответствующий ему индекс. В нашем случае необходимо взять максимум по измерению вектора вероятности (не по измерению батчей), то есть измерению 1:

```
# In[15]:
_, index = torch.max(out, dim=1)
index

# Out[15]:
tensor([1])
```

¹ И хотя можно сказать, что модель сомневается в результате (поскольку дает для двух классов вероятности 48 и 52 %), оказывается, что в типичном случае обучение приводит к слишком сильно уверенным результатам моделей. Байесовские нейронные сети могут помочь решить эту проблему, но их обсуждение выходит за рамки данной книги.

Итак, модель говорит, что на изображении — птица. Просто повезло. Но мы адаптировали выходной сигнал нашей модели к текущей задаче классификации. Кроме того, мы запустили нашу модель для входного изображения и проверили, что наше руководство работает. Пора приступить к обучению сети. Как и в предыдущих двух главах, необходимо выбрать функцию потерь для минимизации во время обучения.

7.2.5. Функция потерь для классификации

Мы уже упоминали, что именно функция потерь придает вероятностям смысл. В главах 5 и 6 в качестве функции потерь применялась среднеквадратичная ошибка (MSE). Можно и здесь воспользоваться ею, добиваясь сходимости выходных вероятностей к $[0.0, 1.0]$ и $[1.0, 0.0]$. Однако, если задуматься, становится ясно, что сами по себе эти значения нас не интересуют. Если взглянуть опять на операцию `argmax`, то можно увидеть, что на самом деле нас интересует то, что первая вероятность больше второй для самолетов и наоборот — для птиц. Другими словами, необходимо накладывать штраф на ошибки классификации, а не кропотливо штрафовать все, что не равно в точности 0,0 или 1,0.

В этом случае необходимо максимизировать вероятность, соответствующую истинному классу `out[class_index]`, где `out` — выходной сигнал многомерной логистической функции, а `class_index` — вектор, содержащий 0 для [метки] *airplane* (самолет) и 1 — для *bird* (птица). Эта величина — соответствующая истинному классу вероятность — называется *правдоподобием* (истинность параметров нашей модели при имеющихся данных)¹. Другими словами, нам нужна функция потерь, принимающая очень высокие значения, когда правдоподобие низко: настолько низко, что вероятности альтернативных вариантов выше. И наоборот, потери должны быть низкими, когда правдоподобие данного варианта выше, чем у альтернатив, и мы не хотим заикливаться на доведении вероятности до 1.

Действующая подобным образом функция потерь существует и называется *отрицательной логарифмической функцией правдоподобия* (*negative log likelihood, NLL*). Выражение для ее вычисления выглядит вот так: `NLL = - sum(log(out_i[c_i]))`, где суммирование производится по N примерам данных, а `c_i` — истинный класс примера данных i . Взглянем на рис. 7.10, на котором изображен график NLL как функции от предсказанной вероятности.

Как видно из этого рисунка, NLL стремится к бесконечности, когда модель присваивает данным низкие вероятности, и уменьшается до относительно низких значений, когда вероятности больше 0,5. Учтите, что в качестве входного сигнала NLL принимает вероятности, так что при росте правдоподобия другие вероятности неизбежно уменьшаются.

¹ Краткое определение основных терминов можно найти здесь: *MacKay D. Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003, раздел 2.3.

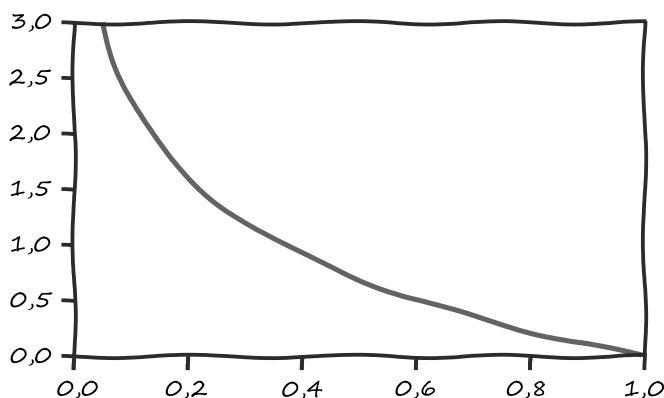


Рис. 7.10. Зависимость функции потерь NLL от предсказанных вероятностей

Подытоживая, нашу функцию потерь для классификации можно вычислить следующим образом. Для каждого примера данных в батче мы делаем следующее.

1. Производим прямой проход и получаем выходные значения из последнего (линейного) слоя.
2. Вычисляем для них многомерную логистическую функцию и получаем вероятности.
3. Извлекаем предсказанную вероятность для истинного класса (правдоподобие параметров). Отметим, что истинный класс известен, поскольку обучение производится с учителем, — это наши эталонные данные.
4. Вычисляем ее логарифм, ставим перед ним знак «минус» и прибавляем к потерям.

Итак, как же проделать вышеописанное в PyTorch? В PyTorch есть класс `nn.NLLLoss`. Впрочем (внимание, подводный камень!), он принимает на входе не вероятности, как можно ожидать, а тензор логарифмов вероятностей. А затем вычисляет для батча данных NLL нашей модели. Для такого соглашения о входных данных есть веские основания: когда вероятность близка к нулю, вычисление ее логарифма связано со сложностями. Выходом из ситуации будет воспользоваться функцией `nn.LogSoftmax` вместо `nn.Softmax`, которая обеспечивает численную устойчивость вычислений.

Доработаем нашу модель, воспользовавшись `nn.LogSoftmax` в качестве выходного модуля:

```
model = nn.Sequential(  
    nn.Linear(3072, 512),  
    nn.Tanh(),  
    nn.Linear(512, 2),  
    nn.LogSoftmax(dim=1))
```

Создаем экземпляр функции NLL-потерь:

```
loss = nn.NLLLoss()
```

Функция потерь получает на входе выходной сигнал `nn.LogSoftmax` с батчем в качестве первого аргумента и тензором индексов классов (нулей и единиц в нашем случае) — в качестве второго. Проверим его на нашей птичке¹:

```
img, label = cifar2[0]

out = model(img.view(-1).unsqueeze(0))

loss(out, torch.tensor([label]))

tensor(0.6509, grad_fn=<NllLossBackward>)
```

В завершение нашего исследования функций потерь взглянем, насколько лучше функция потерь на основе перекрестной энтропии, чем MSE. На рис. 7.11 виден уклон графика функции потерь на основе перекрестной энтропии в том случае, когда предсказание далеко от целевой величины (в углу, соответствующем низкому значению потерь, для истинного класса предсказывается вероятность 99,97 %), в то время как отброшенная нами в самом начале MSE насыщается намного

Удачные и менее удачные функции потерь для классификации

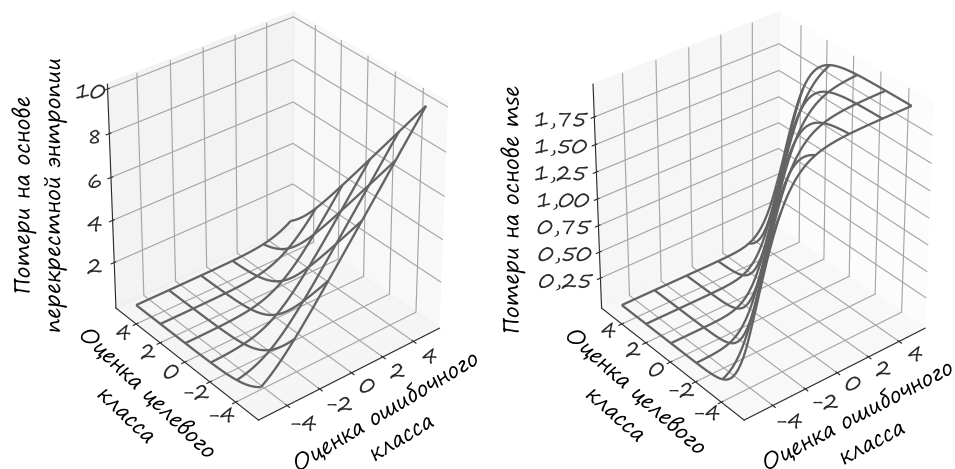


Рис. 7.11. Перекрестная энтропия (слева) и MSE между предсказанными вероятностями и целевым вектором вероятностей (справа) как функции предсказанных потерь, то есть до (логарифмической) многомерной логистической функции

¹ Авторы забыли вставить в листинг комментарии `In[]` и `Out[]`. — Примеч. пер.

раньше и — что принципиально — для совершенно неправильных предсказаний. Основная причина состоит в том, что уклон MSE слишком мал, чтобы компенсировать пологость многомерной логистической функции активации для неправильных предсказаний. Поэтому MSE для вероятностей плохо подходит для задач классификации.

7.2.6. Обучение классификатора

Отлично! Мы готовы вернуться к циклу обучения, написанному нами в главе 5, и посмотреть, как происходит обучение (процесс показан на рис. 7.12)¹:

```
import torch
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1))

learning_rate = 1e-2

optimizer = optim.SGD(model.parameters(), lr=learning_rate)

loss_fn = nn.NLLLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for img, label in cifar2:
        out = model(img.view(-1).unsqueeze(0))
        loss = loss_fn(out, torch.tensor([label]))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

Выводит функцию потерь для последнего изображения. В следующей главе мы внесем усовершенствования и будем получать среднее значение для эпохи в целом

Если присмотреться внимательнее, окажется, что мы кое-что изменили в цикле обучения. В главе 5 цикл был один: по эпохам (напомним, что эпоха завершается после обработки всех примеров данных в обучающем наборе). Мы поняли, что обработка всех 10 000 изображений одним батчем — это перебор, так что решили создать внутренний цикл, чтобы обрабатывать по одному примеру данных за раз и производить обратное распространение ошибки по этому одному примеру.

¹ Для корректной работы этого кода необходимо добавить еще импорт:

```
import torch.optim as optim
```

Примеч. пер.

Ⓐ Для n эпох:

Для каждого примера данных в наборе:
 Вычисляем модель (прямой проход)
 Вычисляем функцию потерь
 Накапливаем градиент
 (обратный проход)
 Обновляем модель на основе
 накопленного градиента

Ⓑ Для n эпох:

Для каждого примера данных в наборе:
 Вычисляем модель (прямой проход)
 Вычисляем функцию потерь
 Накапливаем градиент
 (обратный проход)
 Обновляем модель на основе
 накопленного градиента

Ⓒ Для n эпох:

Разбиваем набор данных
 на мини-батчи
 Для каждого мини-батча:
 Для каждого примера данных в мини-батче
 Вычисляем модель (прямой проход)
 Вычисляем функцию потерь
 Накапливаем градиент
 (обратный проход)
 Обновляем модель на основе градиента



Рис. 7.12. Циклы обучения: А — усреднение обновлений по всему набору данных; В — обновление модели на каждом примере данных; С — усреднение обновлений по мини-батчам

В то время как в первом случае градиент накапливается по всем примерам данных перед применением, в этом случае мы применяем изменения к параметрам на основе очень неполной оценки градиента, основанной на одном примере. Однако направление, подходящее для снижения потерь на основе одного примера, может оказаться неподходящим для других. Перетасовывая примеры данных на каждой эпохе и вычисляя градиент по одному или (что желательно из соображений устойчивости) нескольким примерам данных за раз, мы фактически вносим элемент случайности в алгоритм градиентного спуска. Помните SGD? Эта аббревиатура расшифровывается как *стохастический градиентный спуск*, и S связан именно с этим: с обработкой по небольшим батчам (мини-батчам) перетасованных данных. Оказывается, что следование градиентам, вычисленным по мини-батчам, которые представляют собой лишь слабые аппроксимации градиентов, вычисленных по всему набору данных, улучшает сходимость и предотвращает «застывание» процесса оптимизации во встреченных по пути локальных минимумах. Как показано на рис. 7.13, полученные из мини-батчей градиенты случайным образом отклоняются от идеального направления, и именно поэтому, в частности, желательно использовать относительно малую скорость обучения. Благодаря перетасовке набора данных на каждой эпохе улучшается представительность последовательности вычисленных по мини-батчам градиентов относительно тех градиентов, которые были вычислены по всему набору данных.



Рис. 7.13. Градиентный спуск с усреднением по всему набору данных (бледная линия), по сравнению со стохастическим градиентным спуском, где градиент вычисляется по выбираемым случайным образом мини-батчам

Обычно размер мини-батчей представляет собой константу, задаваемую до обучения, аналогично скорости обучения. Чтобы не путать с параметрами модели, их называют *гиперпараметрами*.

В нашем коде обучения размер мини-батчей составляет 1 — мы выбираем по одному элементу из набора данных за раз. Модуль `torch.utils.data` включает класс, помогающий с перетасовкой и организацией данных по мини-батчам: `DataLoader`. Задача загрузчика данных состоит в выборе мини-батчей из набора данных с гибкими возможностями использования различных стратегий выборки. Одна из самых распространенных стратегий: равномерная выборка после перетасовки данных в каждой эпохе. На рис. 7.14 показан загрузчик данных, перетасовывающий полученные от `Dataset` индексы.

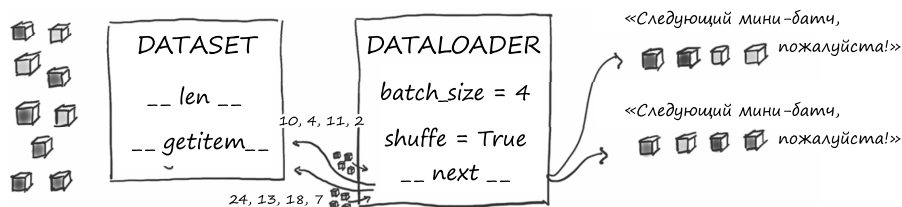


Рис. 7.14. Загрузчик данных, распределяющий данные по мини-батчам, применяя объект `Dataset` для выборки отдельных элементов данных

Взглянем, как это происходит. Как минимум конструктору класса `DataLoader` необходимо передать объект `Dataset`, а также аргумент `batch_size` и булево значение `shuffle`, указывающее, необходимо ли перетасовывать данные в начале каждой эпохи:

```
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)
```

По объекту `DataLoader` можно проходить в цикле, поэтому мы можем использовать его непосредственно во внутреннем цикле нашего нового кода обучения:

```
import torch
import torch.nn as nn

train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1))

learning_rate = 1e-2

optimizer = optim.SGD(model.parameters(), lr=learning_rate)

loss_fn = nn.NLLLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        loss = loss_fn(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

Из-за перетасовки теперь выводится потеря для случайного батча — очевидно, нам нужно улучшить это в главе 8

На каждой итерации внутреннего цикла `imgs` представляет собой тензор размером $64 \times 3 \times 32 \times 32$, то есть мини-батч из 64 RGB-изображений (размером 32×32), а `labels` — тензор размером 64 с индексами меток.

Запускаем обучение:

```
Epoch: 0, Loss: 0.523478
Epoch: 1, Loss: 0.391083
Epoch: 2, Loss: 0.407412
Epoch: 3, Loss: 0.364203
```

```
...
Epoch: 96, Loss: 0.019537
Epoch: 97, Loss: 0.008973
Epoch: 98, Loss: 0.002607
Epoch: 99, Loss: 0.026200
```

Мы видим, что функция потерь убывает, но мы не знаем, достаточно ли низко ее значение. А поскольку наша цель тут в том, чтобы правильно присвоить изображениям метки классов, причем желательно на независимом наборе данных, мы можем вычислить безошибочность нашей модели на проверочном наборе данных в контексте отношения количества правильных классификаций к общему их числу:

```
val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=64,
                                         shuffle=False)

correct = 0
total = 0

with torch.no_grad():
    for imgs, labels in val_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        _, predicted = torch.max(outputs, dim=1)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())

print("Accuracy: %f", correct / total)

Accuracy: 0.794000
```

Неидеально, но намного лучше, чем гадание наугад. В нашу защиту можно сказать, что наша модель была достаточно неглубоким классификатором; удивительно, что она вообще работает. А работает она потому, что набор данных очень прост: множество примеров данных двух классов, скорее всего, с формальными отличиями (например, цвет фона), помогающими модели отличать птиц от самолетов по всего лишь нескольким пикселям.

Безусловно, можно «приукрасить» модель, включив в нее дополнительные слои и повысив таким образом ее глубину и разрешающие возможности. Вот один из возможных вариантов:

```
model = nn.Sequential(
    nn.Linear(3072, 1024),
    nn.Tanh(),
    nn.Linear(1024, 512),
    nn.Tanh(),
    nn.Linear(512, 128),
    nn.Tanh(),
    nn.Linear(128, 2),
    nn.LogSoftmax(dim=1))
```

Здесь мы пытаемся аккуратно сократить количество признаков по направлению к выходному слою в надежде, что промежуточные слои сумеют лучше сжать информацию во все более короткие промежуточные выходные сигналы.

Сочетание `nn.LogSoftmax` и `nn.NLLLoss` эквивалентно `nn.CrossEntropyLoss`. Эта терминология является особенностью PyTorch, поскольку `nn.NLLLoss` вычисляет, по сути, перекрестную энтропию, но с предсказаниями логарифмов вероятности в качестве входного сигнала, а `nn.CrossEntropyLoss` вычисляет оценки (иногда называемые *логитами* (*logits*)). Формально `nn.NLLLoss` представляет собой перекрестную энтропию между распределением Дирака, в котором основной вес придается целевой величине и предсказанному распределению, определяемому входными логарифмическими вероятностями.

Еще большую неразбериху вызывает то обстоятельство, что в теории информации, с точностью до нормализации по размеру выборки, эту перекрестную энтропию можно интерпретировать как отрицательное логарифмическое правдоподобие предсказанного распределения с целевым распределением в качестве исхода. Таким образом, обе функции потерь являются отрицательными логарифмическими функциями правдоподобия параметров модели для конкретных данных при предсказании нашей моделью вероятностей (к которым применена многомерная логистическая функция). В нашей книге эти нюансы неважны, но если эти термины встретятся вам в литературе — не позволяйте наименованиям PyTorch запутать вас.

Достаточно часто последний слой `nn.LogSoftmax` не включают в сеть, используя в качестве функции потерь `nn.CrossEntropyLoss`. Попробуем этот вариант:

```
model = nn.Sequential(
    nn.Linear(3072, 1024),
    nn.Tanh(),
    nn.Linear(1024, 512),
    nn.Tanh(),
    nn.Linear(512, 128),
    nn.Tanh(),
    nn.Linear(128, 2))

loss_fn = nn.CrossEntropyLoss()
```

Обратите внимание, что числа будут *точно такими же*, как и для `nn.LogSoftmax` и `nn.NLLLoss`. Просто удобнее делать все за один проход, при этом единственная проблема будет в том, что выходной сигнал модели нельзя будет интерпретировать как вероятности (или логарифмы вероятностей). Для их получения придется явным образом пропустить выходной сигнал через многомерную логистическую функцию.

Обучив модель и вычислив степень безошибочности на проверочном наборе данных (0,802 000), мы видим, что более крупная модель дает прирост безошибочности, но не настолько уж сильный. Безошибочность на обучающем наборе

данных практически идеальна (0,998 100). О чем эти числа нам говорят? О том, что в обоих случаях модель переобучена. Наша полносвязная модель обучается различать птиц и самолеты на обучающем наборе данных, просто запоминая обучающие примеры, но качество ее работы на проверочном наборе данных будет не слишком хорошим, даже если выбрать модель покрупнее.

PyTorch позволяет быстро выяснить, сколько параметров у модели, с помощью метода `parameters()` объекта `nn.Model` (тот же самый метод, с помощью которого мы передавали параметры оптимизатору). Чтобы узнать, сколько элементов в каждом из экземпляров тензоров, можно вызвать метод `numel`. Их суммирование покажет нам общее количество. В зависимости от сценария использования, при подсчете параметров может потребоваться проверить, установлен ли параметр `requires_grad` в `True`, чтобы отделить количество *обучаемых* параметров от общего размера модели. Взглянем на нашу текущую ситуацию:

```
# In[7]:
numel_list = [p.numel()
               for p in model.parameters()
               if p.requires_grad == True]
sum(numel_list), numel_list

# Out[7]:
(3737474, [3145728, 1024, 524288, 512, 65536, 128, 256, 2])
```

Ничего себе, 3,7 миллиона параметров! Немаленькая сеть для такого маленького входного изображения, правда? Даже наша первая сеть¹ была довольно велика:

```
# In[9]:
numel_list = [p.numel() for p in first_model.parameters()]
sum(numel_list), numel_list

# Out[9]:
(1574402, [1572864, 512, 1024, 2])
```

Количество параметров нашей первой модели примерно вдвое меньше, чем последней. Из списка размеров отдельных параметров становится понятно, кто в этом виноват: первый модуль, включающий 1,5 миллиона параметров. В нашей полносвязной сети 1024 входных признака, из-за чего первый линейный модуль насчитывает 3 миллиона параметров. Этого можно было ожидать: мы знаем, что линейный слой вычисляет $y = \text{weight} * x + \text{bias}$, и если длина x равна 3072 (пренебрегая измерением батчей для простоты), а длина y равна 1024, то размер тензора `weight` должен быть 1024×3072 , а `bias` — 1024. А $1024 \times 3072 + 1024 = 3\,146\,752$, как мы выяснили ранее. Мы можем проверить эти значения напрямую:

```
# In[10]:
linear = nn.Linear(3072, 1024)
```

¹ Из подраздела 7.2.2. — *Примеч. пер.*

```
linear.weight.shape, linear.bias.shape

# Out[10]:
(torch.Size([1024, 3072]), torch.Size([1024]))
```

О чем это нам говорит? О том, что наша нейронная сеть плохо масштабируется при росте количества пикселей. Что произойдет, если у нас будет RGB-изображение 1024×1024 ? Получится 3,1 миллиона входных значений. Даже в случае 1024 скрытых признаков (что для нашего классификатора не подойдет) получится более 3 миллиардов параметров. При использовании 32-битных значений float нам сразу понадобится 12 Гбайт оперативной памяти, и это мы даже еще не затронули второй слой, не говоря уже о вычислении и хранении градиентов. В большинстве современных GPU такая модель попросту не поместится.

7.2.7. Ограничения, накладываемые полносвязностью

Давайте обсудим, что влечет за собой использование линейного модуля для одномерного представления нашего изображения. Происходящее изображено на рис. 7.15. Это напоминает вычисление линейной комбинации каждого входного значения — то есть каждого компонента нашего RGB-изображения — со всеми

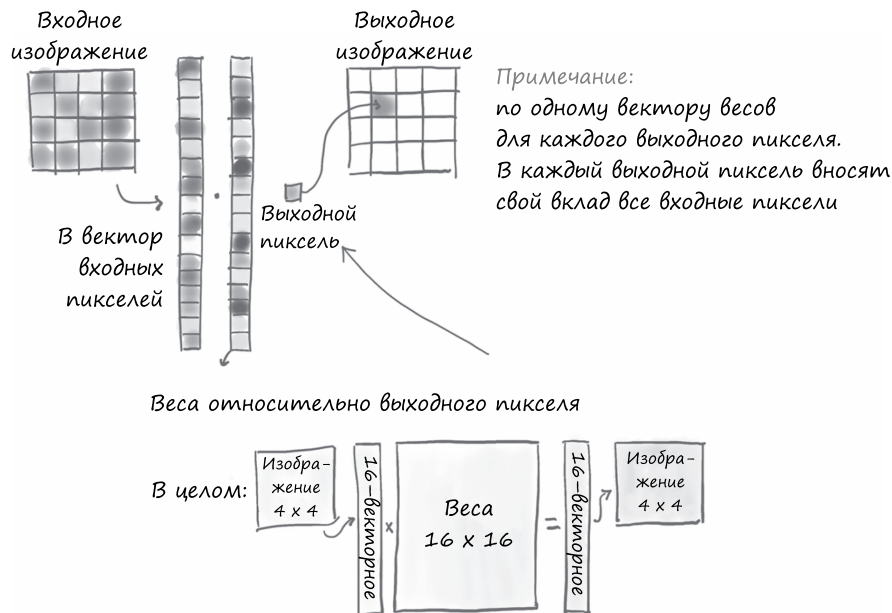
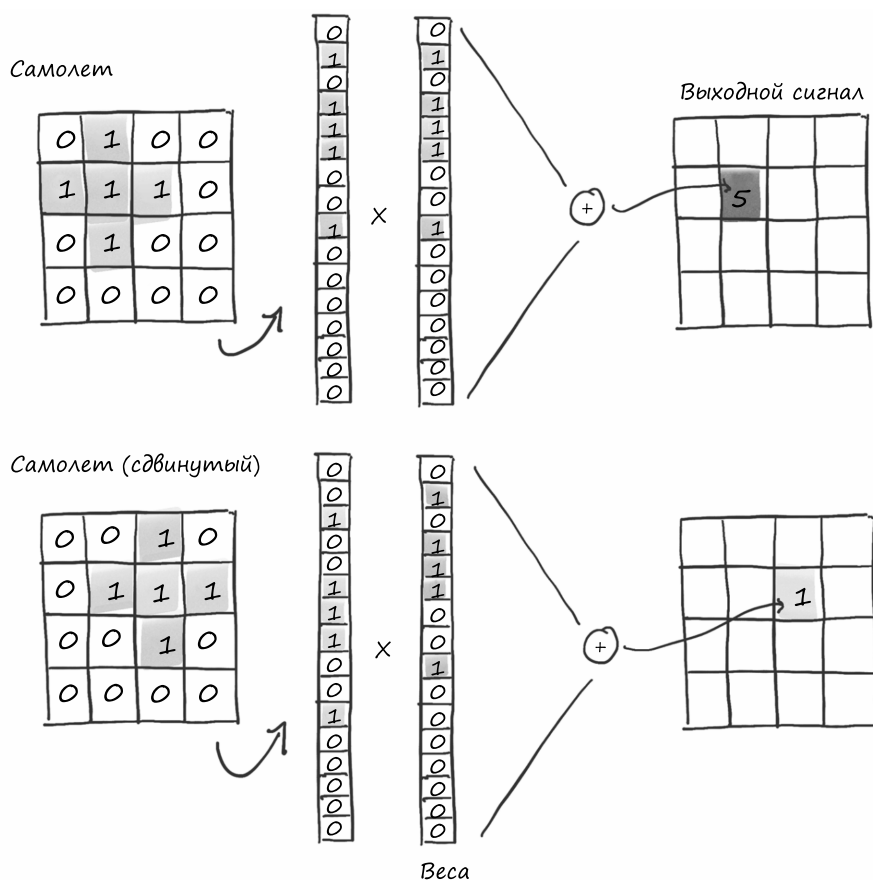


Рис. 7.15. Полносвязный модуль и входное изображение: каждый входной пиксель сочетается с каждым из остальных, в результате чего получаются все элементы выходного сигнала

остальными значениями для каждого выходного признака. С одной стороны, учитывается, что сочетание каждого из пикселей с любым другим пикселем в изображении может потенциально оказаться важным для нашей задачи. С другой стороны, мы не учитываем относительного местоположения соседних или удаленных друг от друга пикселей, поскольку рассматриваем изображение как один большой числовой вектор.

Летающий по небу самолет, захваченный в изображении размером 32×32 , отделенно напоминает темный крестик на синем фоне. Полносвязная сеть, такая как на рис. 7.15, должна усвоить: то, что пиксель 0,1 — темного цвета, пиксель 1,1 — также темного цвета и т. д., указывает на самолет. Все это показано в верхней части рис. 7.16. Однако если сдвинуть тот же самый самолет на один пиксель



или более, как в нижней части рисунка, то модели придется заново усваивать взаимосвязь между пикселями с нуля: на этот раз на метку самолета указывает то, что пиксель 0,2 — темного цвета, пиксель 1,2 — темного цвета и т. д. На более формальном языке полносвязная сеть не является *инвариантной относительно сдвига* (*translation invariant*). Это значит, что сеть, обученная распознавать «Спитфайр»¹, начинающийся с позиции 4,4, не сможет распознать *тот же самый* «Спитфайр», начинающийся с позиции 8,8. Нам пришлось бы *дополнять* (*augment*) набор данных, то есть применять случайные сдвиги к изображениям во время обучения, чтобы сеть могла заметить «Спитфайр» в любом месте изображения, причем это пришлось бы делать для всех изображений в наборе (отметим, что для этой цели мы могли бы присоединить к модели соответствующее преобразование из `torchvision.transforms`). Однако подобное дополнение данных обходится недешево: количество скрытых признаков, то есть параметров, должно быть достаточно большим, чтобы хранить информацию обо всех этих сдвинутых копиях изображений.

Итак, к концу данной главы у нас есть набор данных, модель и цикл обучения, причем наша модель способна обучаться. Однако поскольку структура нашей сети не соответствует поставленной задаче, модель излишне приспосабливается к обучающему набору данных, а не усваивает обобщенные признаки того, что должна обнаруживать.

Мы создали модель, способную связывать каждый из пикселей с любым другим пикселем изображения, вне зависимости от их пространственного расположения. Впрочем, мы сделали обоснованное предположение, что расположенные ближе друг к другу пиксели теоретически намного сильнее связаны. Это значит, что мы обучаем не инвариантный относительно сдвига классификатор, так что нам приходится расходовать немало разрешающих возможностей модели на усвоение сдвинутых копий изображений, чтобы можно было надеяться на хорошие результаты на проверочном наборе данных. Должен же существовать способ лучше, правда?

Конечно, большинство таких вопросов в подобной книге — риторические. Для решения нашего текущего набора задач необходимо внести изменения в модель, добавив сверточные слои. Мы обсудим, что это такое, в следующей главе.

7.3. ИТОГИ ГЛАВЫ

В этой главе мы решили простую задачу классификации от набора данных до модели и до минимизации соответствующей функции потерь в цикле обучения.

¹ Британский истребитель времен Второй мировой войны. — *Примеч. пер.*

Все это станет стандартными инструментами в вашем наборе инструментов PyTorch, а необходимые для их использования навыки пригодятся на протяжении всей вашей карьеры разработчика PyTorch.

Мы также обнаружили серьезный недостаток нашей модели: мы рассматривали двумерные изображения как одномерные данные. Кроме того, у нас пока что нет естественного способа сделать модель инвариантной относительно сдвига. В следующей главе вы узнаете, как воспользоваться двумерностью данных изображений для получения лучших результатов¹.

Мы можем воспользоваться изученным прямо сейчас для обработки данных и без этой инвариантности относительно сдвига. Например, мы могли бы уже сейчас добиться прекрасных результатов для табличных данных или данных временных рядов, с которыми мы сталкивались в главе 4. А в известной мере и для должным образом представленных текстовых данных тоже².

7.4. УПРАЖНЕНИЯ

1. Реализуйте с помощью `torchvision` случайную обрезку изображений.
 - А. Чем полученные изображения отличаются от необрезанных оригиналов?
 - Б. Что будет, если запросить то же изображение во второй раз?
 - В. Что получится в результате обучения на обрезанных случайным образом изображениях?
2. Воспользуйтесь другой функцией потерь (например, MSE).
 - А. Как изменится поведение при обучении?
3. Можно ли сократить разрешающие возможности сети так, чтобы она перестала переобучаться?
 - А. Как при этом ведет себя модель на проверочном множестве?

¹ Та же оговорка относительно инвариантности по сдвигу применима и к чисто одномерным данным: классификатор аудио, вероятно, будет выдавать те же результаты, если классифицируемый звуковой файл начинается на десятую долю секунды раньше или позже.

² С помощью архитектуры сети из этой главы можно работать с моделями мультимножеств слов (bag-of-words models), которые просто усредняют вложения слов. Более современные модели учитывают позиции слов и нуждаются в более продвинутых архитектурах.

7.5. РЕЗЮМЕ

- Машинное зрение — одна из самых обширных сфер применения глубокого обучения.
- Существует несколько общедоступных наборов изображений с описаниями; ко многим из них можно получить доступ через `torchvision`.
- Классы `Dataset` и `DataLoader` — это простые, но эффективные абстракции для загрузки наборов данных и выборки из них данных.
- Для задач классификации можно получить значения, которые могут восприниматься как вероятности с помощью многомерной логистической функции на выходе сети. Идеальная функция потерь для классификации в этом случае получается путем подачи выходного сигнала многомерной логистической функции на вход неотрицательной логарифмической функции правдоподобия. В PyTorch сочетание многомерной логистической функции и подобной функции потерь называется перекрестной энтропией.
- Ничто не мешает нам обрабатывать изображения как векторы значений пикселей с помощью полносвязной сети, подобно любым другим числовым данным. Однако при этом намного сложнее извлечь выгоду из пространственных взаимосвязей в данных.
- Создавать простые модели можно с помощью `nn.Sequential`.

8

Обобщение с помощью сверток

В этой главе

- ✓ Свертки.
- ✓ Создание сверточной нейронной сети.
- ✓ Создание пользовательских подклассов `nn.Module`.
- ✓ Разница между модулем и функциональным API.
- ✓ Варианты архитектуры нейронных сетей.

В предыдущей главе мы создали простую нейронную сеть, обучающуюся (или переобучающуюся) на данных благодаря большому числу доступных для оптимизации параметров в линейных слоях. Впрочем, у нас были проблемы с нашей моделью, поскольку она запоминала обучающий набор данных лучше, чем обобщала характеристики птиц и самолетов. Благодаря анализу архитектуры модели у нас появилось предположение, почему так могло происходить. Из-за полносвязной архитектуры, необходимой для обнаружения различных возможных сдвигов птиц или самолетов на изображении, мы имеем слишком много параметров (что облегчает запоминание моделью обучающего набора) и зависимость от положения (что усложнит обобщение). Как мы обсуждали в предыдущей главе, обучающие данные можно дополнить большим спектром перекадрированных изображений, чтобы попытаться добиться обобщения на новых данных, но это не решит проблему слишком большого количества параметров.

Существует способ получше! Он заключается в замене плотного, полносвязного аффинного преобразования в нейроне нашей сети другой линейной операцией: сверткой.

8.1. АРГУМЕНТЫ В ПОЛЬЗУ СВЕРТОК

Давайте докапемся до самой сути того, что такое свертки и как их использовать в нейронных сетях. Да-да, сейчас мы на полпути к поиску решения задачи распознавания птиц и самолетов, но это отступление стоит того, чтобы потратить на него время. Мы научимся интуитивно понимать эту основополагающую концепцию машинного зрения, после чего с новыми суперспособностями вернемся к нашей задаче.

В этом разделе мы увидим, как свертки обеспечивают локальность и инвариантность относительно сдвига. Для этого мы внимательно изучим формулу, описывающую свертки, и применим ее на бумаге, но не волнуйтесь, главное будет на рисунках, а не в формулах.

Мы уже упоминали, что умножение одномерного представления нашего входного изображения на матрицу весов $n_{\text{выходных_признаков}} \times n_{\text{входных_признаков}}$, производимое в `nn.Linear`, означает вычисление для каждого канала изображения взвешенной суммы всех пикселей, умноженной на множество весов, по одному на выходной признак. Мы также говорили, что если нужно распознавать соответствующие объектам, например самолетам в небе, закономерности, вероятно, понадобится проанализировать взаимное расположение близлежащих пикселей, а сочетания более удаленных пикселей нас будет интересовать меньше. Фактически неважно, есть ли в углу нашего изображения «Спитфайра» дерево, облако или воздушный змей.

Чтобы математически выразить наши интуитивные знания, можно вычислить взвешенную сумму пикселя только с его непосредственными соседями, а не со всеми прочими пикселями изображения. Это эквивалентно созданию весовых матриц, по одной для каждого выходного признака и местоположения выходного пикселя, в которых все веса на определенном расстоянии от центрального пикселя равны нулю. Такая сумма все равно будет взвешенной, то есть линейной операцией.

8.1.1. Что делают свертки

Мы уже указали ранее на одно желательное свойство: эти локализованные закономерности должны влиять на выходной сигнал вне зависимости от их местоположения на изображении, то есть обеспечивать *инвариантность относительно сдвига*. Чтобы добиться этого в использовавшейся в главе 7 матрице, применяемой к изображению в виде вектора, понадобилось бы реализовать довольно сложную закономерность весов (не волнуйтесь, если она покажется

вам *слишком* сложной; скоро мы все усовершенствуем): большая часть весовой матрицы должна была бы содержать нули (для записей, соответствующих входным пикселям, расположенным слишком далеко от выходного, чтобы на него как-то повлиять). Для остальных весовых коэффициентов пришлось бы как-то согласовывать записи, соответствующие одной относительной позиции входного и выходного пикселей. Это значит, что необходимо задать для них одинаковые начальные значения и гарантировать, что все эти *связанные* весовые коэффициенты остаются одинаковыми при обновлении сети во время обучения. Таким образом мы гарантируем, что весовые коэффициенты реагируют на локальные закономерности в неких окрестностях, причем эти локальные закономерности распознаются независимо от того, где они встречаются на изображении.

Конечно, такой подход совершенно непригоден для использования на практике. К счастью, существует вполне доступная, локальная, инвариантная относительно сдвига линейная операция над изображениями: *свертка* (*convolution*). Можно описать свертку и более лаконично, но то, что далее описано, хотя и с несколько нестандартной точки зрения, обладает всеми упомянутыми свойствами.

Свертка или, точнее, *дискретная свертка* (*discrete convolution*)¹, потому что существует и аналогичный непрерывный вариант свертки, который мы тут не будем обсуждать, определяется для двумерного изображения как скалярное произведение матрицы весовых коэффициентов — *ядра* (*kernel*) — на каждую окрестность входных данных. Рассмотрим ядро 3×3 (в глубоком обучении обычно применяются маленькие ядра; далее мы увидим почему), то есть двумерный тензор:

```
weight = torch.tensor([[w00, w01, w02],
                        [w10, w11, w12],
                        [w20, w21, w22]])
```

и одноканальное изображение $M \times N$:

```
image = torch.tensor([[i00, i01, i02, i03, ..., i0N],
                       [i10, i11, i12, i13, ..., i1N],
                       [i20, i21, i22, i23, ..., i2N],
                       [i30, i31, i32, i33, ..., i3N],
                       ...
                       [iM0, iM1, iM2, iM3, ..., iMN]])
```

Элемент выходного изображения (без смещения) можно вычислить следующим образом:

```
o11 = i11 * w00 + i12 * w01 + i13 * w02 +
       i21 * w10 + i22 * w11 + i23 * w12 +
       i31 * w20 + i32 * w21 + i33 * w22
```

¹ Существует небольшое различие между сверткой в PyTorch и математической операцией свертки: знак одного аргумента — противоположный. Если немного «позанудствовать», мы могли бы назвать свертку PyTorch дискретной взаимной корреляцией.

На рис. 8.1 показаны эти вычисления.

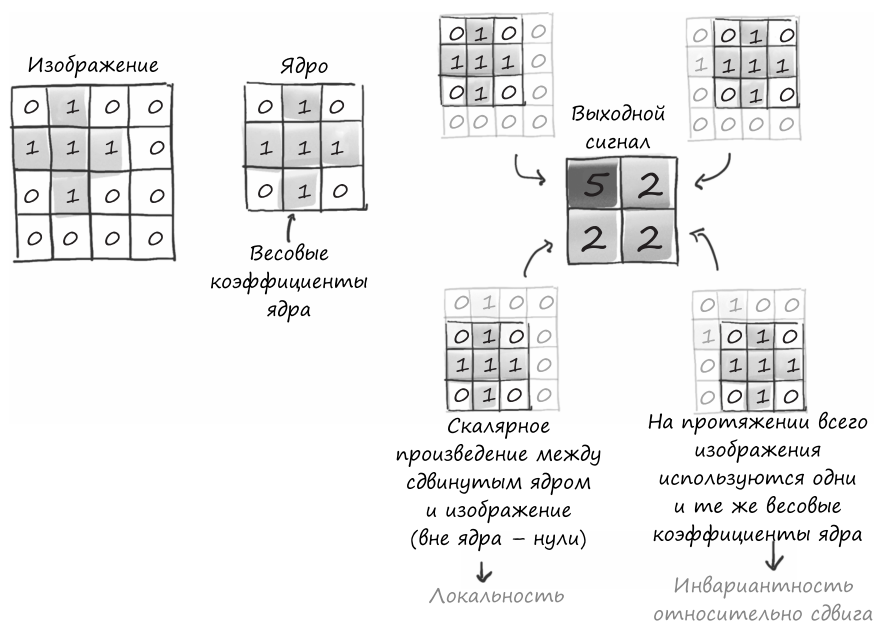


Рис. 8.1. Свертка: локальность и инвариантность относительно сдвига

Другими словами, мы «сдвигаем» ядро на позицию **i11** входного изображения и умножаем каждый из весовых коэффициентов на значение из соответствующего места входного изображения. Таким образом, выходное изображение получается путем сдвига ядра по всем позициям входного изображения и вычисления взвешенной суммы. В случае многоканального изображения, например для нашего RGB-изображения, матрица весов будет иметь форму $3 \times 3 \times 3$: для каждого канала по одному набору весов, вносящему свой вклад в выходные значения.

Учтите, что, подобно элементам матрицы `weight` объекта `nn.Linear`, весовые коэффициенты ядра заранее не известны, а инициализируются случайным образом и обновляются посредством обратного распространения ошибки. Отметим также, что для всего изображения используется одно и то же ядро, а это значит, что и весовые коэффициенты ядра. Если вернуться к автоматическому вычислению градиентов, это значит, что у каждого весового коэффициента есть история, охватывающая все изображение. Следовательно, в производную функции потерь по сверточным весам вносит свой вклад все изображение.

Теперь нам ясна связь с тем, о чем мы говорили ранее: свертка эквивалентна нескольким линейным операциям, весовые коэффициенты которых равны нулю практически везде, кроме окрестностей отдельных пикселей, и получают одинаковые обновления во время обучения.

В итоге благодаря переходу на свертки мы получаем:

- локальные операции над окрестностями отдельных пикселей;
- инвариантность относительно сдвига;
- уменьшение числа параметров моделей.

Главное в третьем пункте то, что при использовании сверточного слоя количество параметров зависит не от числа пикселей в изображении, как в случае полносвязной модели, а от размера ядра свертки (3×3 , 5×5 и т. д.), а также от числа сверточных фильтров (или выходных каналов) модели.

8.2. СВЕРТКИ В ДЕЙСТВИИ

Похоже, мы потратили уже достаточно времени на теорию! Давайте посмотрим на PyTorch в действии на нашей задаче с птичками и самолетами. Модуль `torch.nn` позволяет производить свертки в одном, двух и трех измерениях: `nn.Conv1d` для временных рядов, `nn.Conv2d` для изображений и `nn.Conv3d` для объемных пространственных данных или видеоданных.

Для нашего набора данных CIFAR-10 подойдет `nn.Conv2d`. В качестве аргументов для `nn.Conv2d` необходимо указать как минимум число входных признаков (или *каналов*, поскольку наши изображения — многоканальные, то есть содержат более одного значения на пиксель), число выходных признаков и размер ядра. Например, у нас три входных признака на пиксель (канала RGB) для первого сверточного модуля и произвольного количества выходных каналов — допустим, 16. Чем больше каналов в выходном изображении, тем больше разрешающие возможности сети. Каналы нужны, чтобы обнаруживать различные типы признаков. Кроме того, поскольку начальные значения для них задаются случайным образом, часть полученных признаков даже после обучения окажутся бесполезными¹. Давайте остановимся на ядре размером 3×3 .

Очень часто применяют ядра, размеры которых одинаковы по всем измерениям, поэтому в PyTorch есть сокращенная форма записи для них: `kernel_size=3` для двумерной свертки означает форму 3×3 (в Python задается в виде кортежа `(3, 3)`), для трехмерной свертки — форму $3 \times 3 \times 3$. У КТ-снимков, которые мы увидим в части II этой книги, разрешение вокселей (объемных пространственных пикселей) по одной из трех осей координат отличается. В противном случае имеет смысл использовать ядра с отдельным размером для этого конкретного измерения. Но пока что мы остановимся на свертках одинакового размера по всем измерениям:

¹ Это часть гипотезы лотерейного билета: что польза от многих из ядер будет не больше чем от проигрышных лотерейных билетов. См.: *Frankle J., Carbin M. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*, 2019, <https://arxiv.org/abs/1803.03635>.

```
# In[11]:
conv = nn.Conv2d(3, 16, kernel_size=3)
conv
```

Вместо сокращенной формы записи `kernel_size=3` мы можем передать кортеж, показанный в выведенных результатах: `kernel_size=(3, 3)`

```
# Out[11]:
Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
```

Какую форму тензора `weight` мы ожидаем? Размер ядра 3×3 , поэтому вес должен состоять из частей размером 3×3 . Для отдельного входного пикселя наше ядро будет рассматривать, допустим, входные каналы `in_ch = 3`, так что форма весового компонента для отдельного значения выходного пикселя (а благодаря инвариантности относительно сдвига — и для всего выходного канала) будет `in_ch \times 3 \times 3`. Наконец, их количество будет равно количеству выходных каналов, в данном случае `out_ch = 16`, так что форма полного тензора весов `out_ch \times in_ch \times 3 \times 3`, то есть $16 \times 3 \times 3 \times 3$ в нашем случае. Размер смещения будет 16 (мы уже давно не упоминали смещение ради простоты, но, как и в случае линейного модуля, оно представляет собой константное значение, прибавляемое к каждому каналу выходного изображения). Проверим наши расчеты:

```
# In[12]:
conv.weight.shape, conv.bias.shape
```

```
# Out[12]:
(torch.Size([16, 3, 3, 3]), torch.Size([16]))
```

Мы видим, что свертки прекрасно подходят для усвоения информации из изображений. Модели становятся меньше и ищут локальные закономерности, весовые коэффициенты которых оптимизированы на основе всего изображения.

В результате прохода двумерной свертки получается двумерное изображение, пиксели которого представляют собой взвешенную сумму значений по локальным окрестностям входного изображения.

В нашем случае как начальные значения весовых коэффициентов ядра `conv.weight`, так и смещения задаются случайным образом, так что выходное изображение особого смысла не несет. Как обычно, необходимо с помощью `unsqueeze` добавить нулевое измерение батчей для вызова модуля `conv` с одним входным изображением, поскольку `nn.Conv2d` ожидает на входе тензор формы $B \times C \times H \times W$:

```
# In[13]:
img, _ = cifar2[0]
output = conv(img.unsqueeze(0))
img.unsqueeze(0).shape, output.shape
```

```
# Out[13]:
(torch.Size([1, 3, 32, 32]), torch.Size([1, 16, 30, 30]))
```

Интересно будет посмотреть на `output`, показанный на рис. 8.2:

```
# In[15]:
plt.imshow(output[0, 0].detach(), cmap='gray')
plt.show()
```

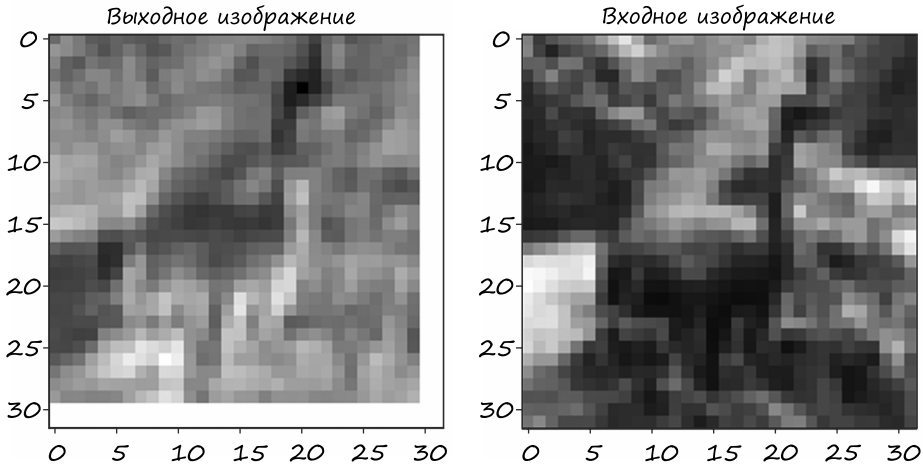


Рис. 8.2. Наша птица после применения случайного сверточного преобразования (мы немного жульничали с кодом, чтобы показать и входное изображение тоже)

Погодите-ка. Взглянем на размер `output`: `torch.Size([1, 16, 30, 30])`. Кхм, похоже, мы потеряли несколько пикселей по пути. Как же это случилось?

8.2.1. Дополнение нулями по краям

То, что наше выходное изображение меньше входного, является побочным эффектом принятия решения о том, что делать на краях изображения. Применение сверточного ядра в виде взвешенной суммы пикселей в окрестности размера 3×3 требует наличия соседних пикселей во всех направлениях. А в точке `i00` соседние пиксели есть только справа и снизу. По умолчанию сверточное ядро в PyTorch проходит по входному изображению, получая `ширина - ширина_ядра + 1` по горизонтали и вертикали. В случае ядер нечетного размера в результате получаются изображения, которые меньше с каждой стороны на половину ширины сверточного ядра (в нашем случае $3 // 2 = 1$). Поэтому нам и не хватает по два пикселя в каждом измерении.

Впрочем, PyTorch позволяет *дополнять нулями* (*padding*) изображения, путем добавления границ с помощью *фигтивных* (*ghost*) пикселей с нулевым значением. На рис. 8.3 показано, как происходит дополнение нулями.

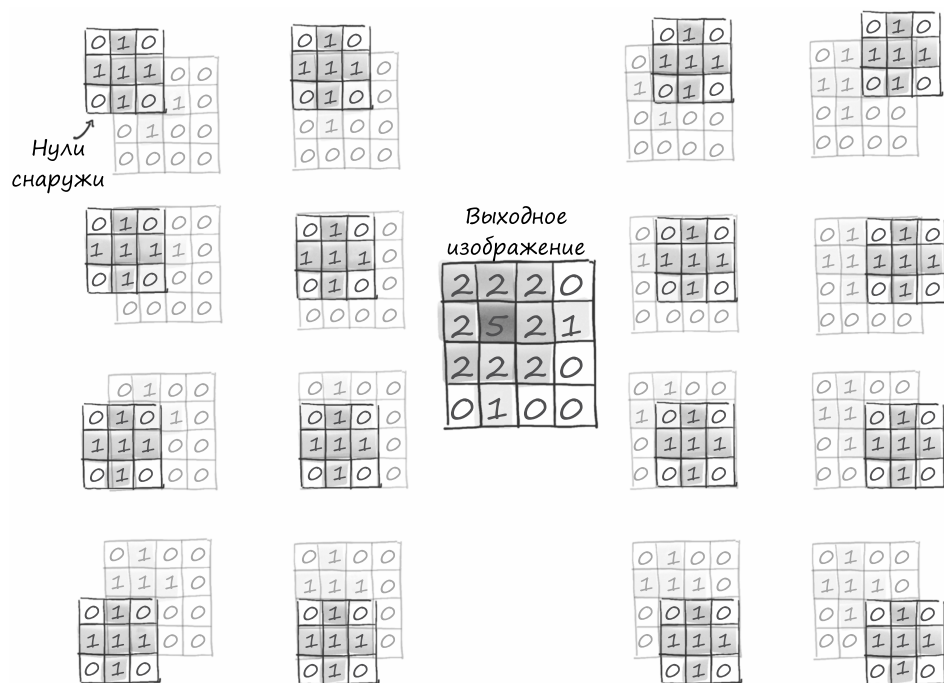


Рис. 8.3. Дополнение нулями ради сохранения размера входного изображения в выходном

В нашем случае параметр `padding=1` при `kernel_size=3` означает появление у `img` набора дополнительных соседей сверху и слева, что позволяет вычислять выходной сигнал свертки даже в углу исходного изображения¹. В итоге размер выходного изображения оказывается точно таким же, как и входного:

```
# In[16]:
conv = nn.Conv2d(3, 1, kernel_size=3, padding=1)  ← Теперь дополнено нулями
output = conv(img.unsqueeze(0))
img.unsqueeze(0).shape, output.shape

# Out[16]:
(torch.Size([1, 3, 32, 32]), torch.Size([1, 1, 32, 32]))
```

Обратите внимание, что размеры `weight` и `bias` не меняются вне зависимости от дополнения.

¹ Для ядер четного размера пришлось бы дополнять другим количеством соседей слева и справа (а также сверху и снизу). PyTorch не предоставляет подобной возможности в самой свертке, но дает возможность решить эту задачу с помощью функции `torch.nn.functional.pad`. Но лучше придерживаться нечетных размеров ядер; ядра четного размера встречаются редко.

Существует две основные причины для дополнения сверток. Во-первых, чтобы разделять задачи выполнения свертки и изменения размера изображения и помнить на одну вещь меньше. Во-вторых, для более изолированных структур, например *обходных связей* (*skip connections*)¹ (обсуждаются в подразделе 8.5.3) и сети U-Nets, которые мы рассмотрим в части II, желательно, чтобы размеры тензоров до и после нескольких сверток оставались совместимыми, чтобы их можно было складывать и вычитать.

8.2.2. Обнаружение признаков с помощью сверток

Мы уже говорили, что `weight` и `bias` — это параметры, усваиваемые посредством обратного распространения ошибки, точно так же, как `weight` и `bias` в `nn.Linear`. Однако можно поэкспериментировать со сверткой, задавая весовые коэффициенты вручную, и посмотреть, что получится.

Сначала обнулим `bias`, просто чтобы исключить все мешающие факторы, после чего зададим константное значение `weight`, чтобы каждый пиксель выходного изображения получал среднее значение своих соседей. Для каждой окрестности 3×3 :

```
# In[17]:
with torch.no_grad():
    conv.bias.zero_()

with torch.no_grad():
    conv.weight.fill_(1.0 / 9.0)
```

Можно было воспользоваться `conv.weight.one_()` — при этом каждый пиксель выходного изображения был бы равен *сумме* окрестных пикселей. Различия незначительны, разве что значения пикселей в выходном изображении оказались бы в девять раз больше.

В любом случае взглянем, как это повлияет на наше изображение из набора CIFAR:

```
# In[18]:
output = conv(img.unsqueeze(0))
plt.imshow(output[0, 0].detach(), cmap='gray')
plt.show()
```

Как можно было предвидеть, фильтр генерирует размытую версию изображения, как показано на рис. 8.4. В конце концов, каждый пиксель выходного изображения представляет собой среднее значение одной из окрестностей входного изображения, так что пиксели выходного изображения связаны друг с другом и меняются более гладко.

¹ В русскоязычной литературе пока нет устоявшегося термина для этого довольно нового понятия. Встречаются такие варианты, как «обходные соединения», «соединения прямого доступа», «пропускаемые соединения». — *Примеч. пер.*

Попробуем что-нибудь другое. На первый взгляд следующее ядро может показаться довольно загадочным:

```
# In[19]:
conv = nn.Conv2d(3, 1, kernel_size=3, padding=1)

with torch.no_grad():
    conv.weight[:] = torch.tensor([[-1.0, 0.0, 1.0],
                                   [-1.0, 0.0, 1.0],
                                   [-1.0, 0.0, 1.0]])

    conv.bias.zero_()
```

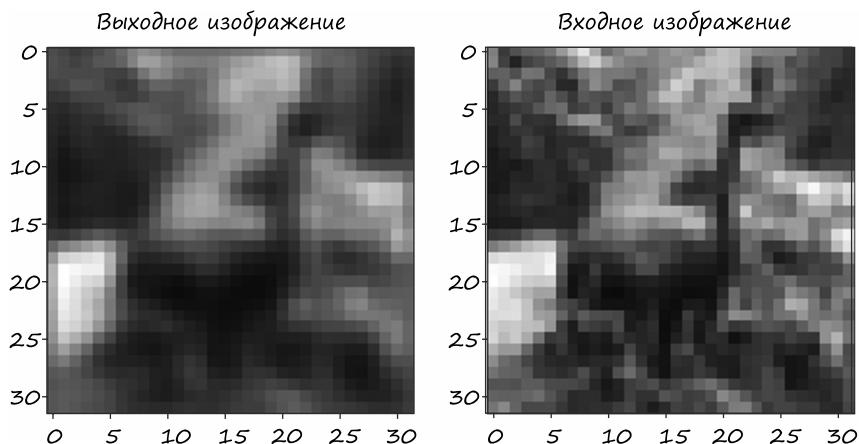


Рис. 8.4. Наша птица, на этот раз размытая благодаря постоянному сверточному ядру

Если записать выражение взвешенной суммы для произвольного пикселя на позиции 2,2, как мы делали ранее для общего сверточного ядра, то мы получим

```
o22 = i13 - i11 +
      i23 - i21 +
      i33 - i31
```

Здесь вычисляется разность всех пикселей справа от i_{22} и пикселей слева от него. В случае применения ядра к вертикальной границе между двумя смежными областями различной яркости значение o_{22} будет выше. При применении же ядра к области равномерной яркости значение o_{22} будет нулевым. Таким образом, наше ядро *выявляет края*, подчеркивая вертикальные края между двумя смежными по горизонтали областями.

Применяя к нашему изображению сверточное ядро, получаем показанный на рис. 8.5 результат. Как и ожидалось, сверточное ядро усиливает вертикальные края. Можно создать намного более изощренные фильтры, например, для обнаружения горизонтальных и диагональных краев, крестовидных или расположенных

в шахматном порядке узоров, где «обнаружение» означает высокие порядки значений в выходном изображении. На самом деле исторически задача специалиста по машинному зрению и заключалась в поиске наиболее эффективного сочетания фильтров для выделения определенных признаков в изображениях и распознавания объектов.

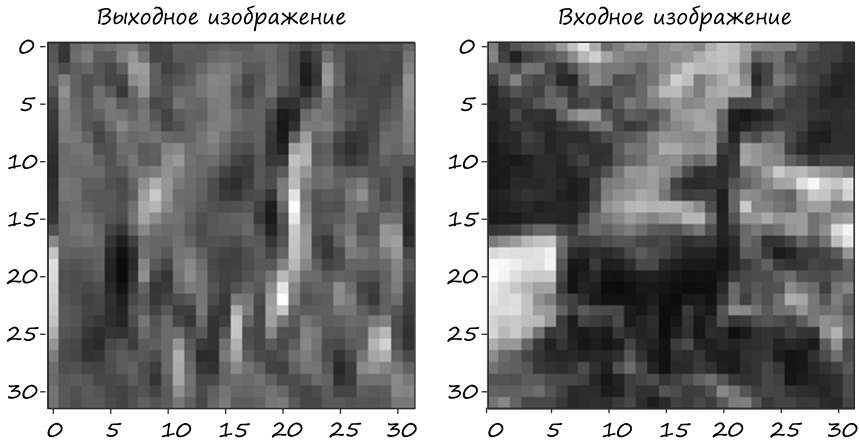


Рис. 8.5. Вертикальные края нашей птицы, полученные благодаря нашему рукотворному сверточному ядру

При использовании глубокого обучения оценка ядер может производиться так, чтобы модель различала как можно эффективнее: например, в контексте минимизации функции потерь на основе отрицательной перекрестной энтропии между выходным сигналом и эталонными данными, с которой мы познакомили вас в подразделе 7.2.5. С этой точки зрения задача сверточной нейронной сети состоит в оценке ядра набора фильтров в последовательных слоях, преобразующих многоканальное изображение в другое многоканальное изображение, в котором различные каналы соответствуют разным признакам (например, один канал — для среднего значения, другой — для вертикальных краев и т. д.). На рис. 8.6 показано, как модель автоматически усваивает ядра.

8.2.3. Расширяем кругозор с помощью субдискретизации и повышения глубины сети

Все это очень хорошо, но остается главный вопрос. Мы обрадовались, что благодаря переходу от полносвязных слоев к сверткам добились локальности и инвариантности относительно сдвига. Далее мы рекомендовали использовать маленькие ядра (3×3 или 5×5), и локальность достигла максимума, это прекрасно. Но как насчет *общей картины*? Откуда мы знаем, что все структуры

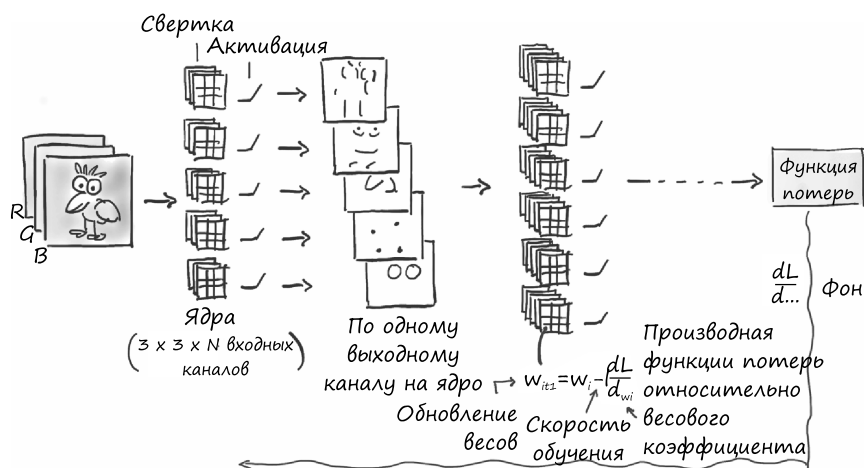


Рис. 8.6. Процесс обучения с помощью сверток путем оценки градиентов относительно весовых коэффициентов ядра и обновления их по отдельности с целью оптимизации функции потерь

в наших изображениях — шириной 3 или 5 пикселей? Что ж, на самом деле не знаем, поскольку это просто не так. А раз это не так, как же наши сети смогут обнаружить закономерности большего масштаба? А это нам точно понадобится для эффективного решения задачи различения птиц от самолетов, поскольку, хотя изображения CIFAR-10 и невелики, размах (крыльев) объектов может достигать двузначного числа пикселей.

Одним из вариантов будет воспользоваться большими сверточными ядрами. Что ж, конечно, в крайнем случае можно использовать ядро 32×32 для изображения 32×32 , но при этом мы вернемся к старому доброму полносвязному аффинному преобразованию и утратим все преимущества свертки. Другой вариант, используемый в сверточных нейронных сетях, заключается в использовании нескольких сверток, одной за другой, с понижающей дискретизацией между ними.

От большого к малому: понижающая дискретизация

Понижающая дискретизация может происходить по-разному. Уменьшение изображения вдвое эквивалентно генерации одного пикселя на выходе на основе четырех соседних пикселей на входе. Как именно вычислять значение на выходе по значениям на входе — наше дело. Можно:

- *усреднять значения четырех входных пикселей* — изначально весьма распространенный подход под названием «*усредняющая субдискретизация*» (*average pooling*), несколько утративший популярность в последнее время;
- *брать максимум из значений четырех входных пикселей* — в настоящее время чаще всего используется именно этот подход: *субдискретизация с выбором*

максимального значения (*max pooling*). Его недостаток в том, что отбрасываются три четверти данных;

- *шаговая свертка (strided convolution)*, при которой учитывается лишь каждый N -й пиксель — свертка 3×4 с шагом 2 позволяет учесть значения всех пикселей предыдущего слоя. Судя по публикациям, этот подход весьма перспективен, хотя и не вытеснил пока что субдискретизацию с выбором максимального значения.

Далее мы сосредоточим свое внимание на субдискретизации с выбором максимального значения, показанной на рис. 8.7. На рисунке приведена наиболее распространенная схема, в которой в качестве значения нового пикселя уменьшенного изображения берется максимум по неперекрывающимся плиткам 2×2 .

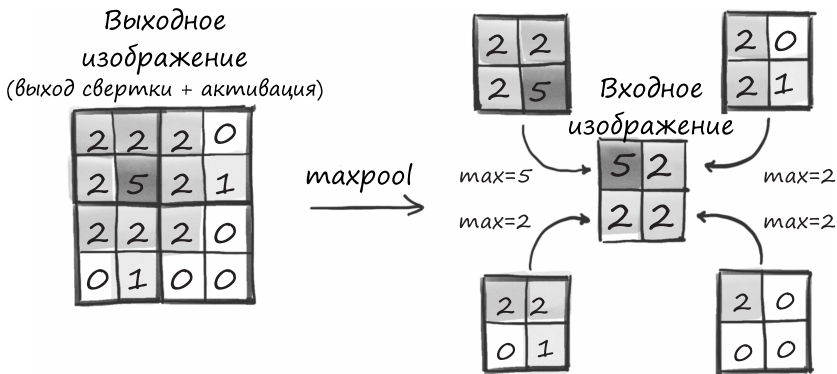


Рис. 8.7. Выбор максимального значения в деталях

Интуитивно понятно, что порядок значений выходных изображений из сверточного слоя, особенно если за ним следует функция активации, как и за любым другим линейным слоем, обычно оказывается довольно большим там, где обнаружены определенные признаки, соответствующие используемому ядру (например, вертикальные линии). Использование в качестве субдискретизированного выходного значения максимума из значений в окрестности размером 2×2 гарантирует *сохранение* обнаруженных признаков в процессе субдискретизации, за счет более слабых сигналов.

Возможности субдискретизации с выбором максимального значения предоставляет модуль `nn.MaxPool2d` (как и для свертки, существуют его версии для одномерных и трехмерных данных). В качестве входного аргумента ему передается размер окрестности для работы операции субдискретизации. Например, для понижающей дискретизации изображения в два раза необходимо передавать значение 2. Давайте проверим, что все работает правильно, прямо на нашем входном изображении:

```
# In[21]:
pool = nn.MaxPool2d(2)
output = pool(img.unsqueeze(0))

img.unsqueeze(0).shape, output.shape

# Out[21]:
(torch.Size([1, 3, 32, 32]), torch.Size([1, 3, 16, 16]))
```

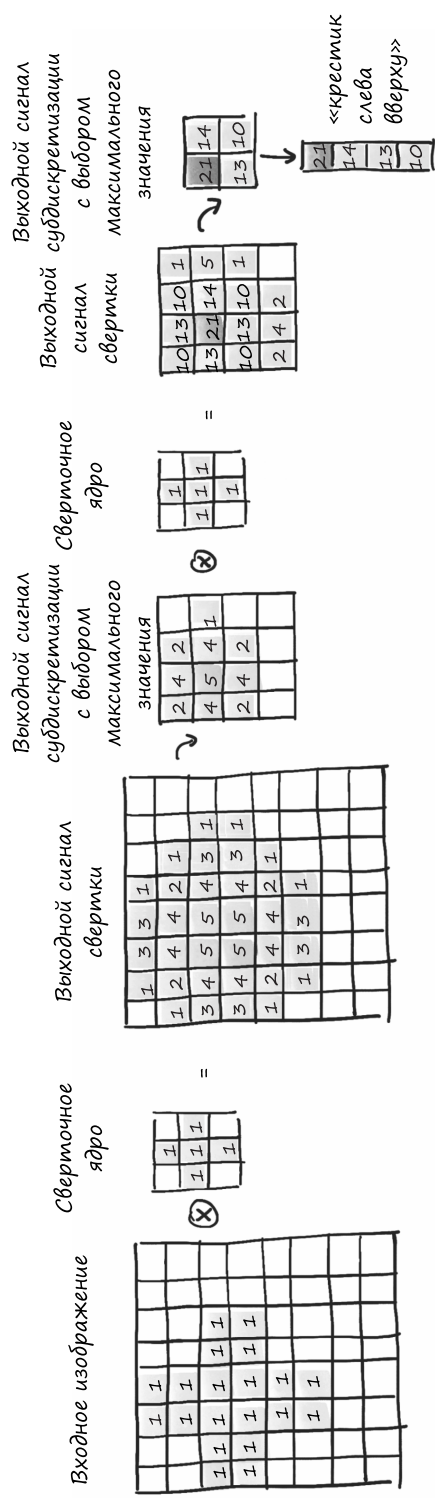
Сочетание сверточных слоев и понижающей дискретизации для лучшего распознавания

Посмотрим теперь, как сочетание сверточных слоев и понижающей дискретизации позволяет распознавать более крупные структуры. На рис. 8.8 мы начинаем с применения набора ядер 3×3 к изображению размера 8×8 , в результате чего получается многоканальное изображение того же размера. Затем мы масштабируем изображение наполовину, получая изображение 4×4 , и применяем к нему еще один набор ядер 3×3 . Этот второй набор ядер работает с окрестностями 3×3 изображения, которое было уменьшено вдвое, фактически соответствующими окрестностям 8×8 исходного изображения. Кроме того, второй набор ядер получает на входе выходной сигнал первого набора ядер (признаки наподобие средних значений, краев и т. д.) и выделяет еще дополнительные признаки.

Итак, первый набор ядер работает с маленькими окрестностями низкоуровневых признаков первого порядка, а второй набор фактически работает с более широкими окрестностями, генерируя признаки, представляющие собой композицию предыдущих признаков. Благодаря этому замечательному механизму сверточные нейронные сети способны анализировать очень сложные кадры — намного сложнее наших изображений 32×32 из набора данных CIFAR-10.

РЕЦЕПТИВНОЕ ПОЛЕ ВЫХОДНЫХ ПИКСЕЛЕЙ

Второе сверточное ядро выдает 21 в качестве выходного сигнала свертки на рис. 8.8 на основе 3×3 верхних левых пикселей первого выходного сигнала субдискретизации с выбором максимального значения. Они, в свою очередь, соответствуют 6×6 пикселям из верхнего левого угла выходного сигнала первой свертки, которые, в свою очередь, вычисляются первой операцией свертки на основе 7×7 верхних левых пикселей. Таким образом, на один пиксель выходного сигнала второй свертки влияет квадрат 7×7 входных пикселей. В первой свертке также для генерации выходного сигнала в углу используются неявно «дополненные нулями» столбец и строка; в противном случае на конкретный пиксель (не у края) выходного сигнала второй свертки влиял бы квадрат 8×8 входных пикселей. Говоря научным языком: данный выходной нейрон конструкции из свертки 3×3 , 2×2 -субдискретизации с выбором максимального значения и свертки 3×3 обладает *рецептивным полем* 8×8 .



8.2.4. Собираем нашу нейронную сеть воедино

Теперь у нас есть все «кирпичики», чтобы создать нашу сверточную нейронную сеть для различения птиц и самолетов. Возьмем в качестве отправной точки нашу предыдущую полносвязную модель и введем в нее вышеупомянутые слои `nn.Conv2d` и `nn.MaxPool2d`:

```
# In[22]:
model = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    nn.Conv2d(16, 8, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    # ...
)
```

Первая операция свертки превращает три канала RGB в 16, благодаря чему у сети появляется возможность генерировать 16 независимых признаков, которые (надемся) позволят различить низкоуровневые признаки птиц и самолетов. Далее мы применяем функцию активации `Tanh`. Полученное 16-канальное изображение 32×32 субдискретизируется первым слоем `nn.MaxPool2d` до 16-канального изображения 16×16 . Теперь субдискретизированное изображение подвергается еще одной операции свертки, выдающей на выходе 8-канальный выходной сигнал 16×16 . Если повезет, это выходное изображение будет состоять из высокоуровневых признаков. И опять же мы применяем функцию активации `Tanh`, после чего производим субдискретизацию до 8-канального выходного изображения 8×8 .

Когда же этот процесс завершается? После уменьшения входного изображения до набора 8×8 признаков можно надеяться вернуть из сети значения вероятностей, подходящих для подачи на вход отрицательной логарифмической функции правдоподобия. Однако вероятности представляют собой пару чисел в одномерном векторе (одно для самолета, одно для птицы), а мы все еще имеем дело с многоканальными двумерными признаками.

Вспоминая начало этой главы, мы уже знаем, что нужно сделать: преобразовать 8-канальное изображение 8×8 в одномерный вектор и завершить нашу сеть набором полносвязных слоев:

```
# In[23]:
model = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    nn.Conv2d(16, 8, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
```

```
# ...
nn.Linear(8 * 8 * 8, 32),
nn.Tanh(),
nn.Linear(32, 2))
```

← Внимание: здесь пропущено нечто важное!

В результате этого кода получается изображенная на рис. 8.9 нейронная сеть.

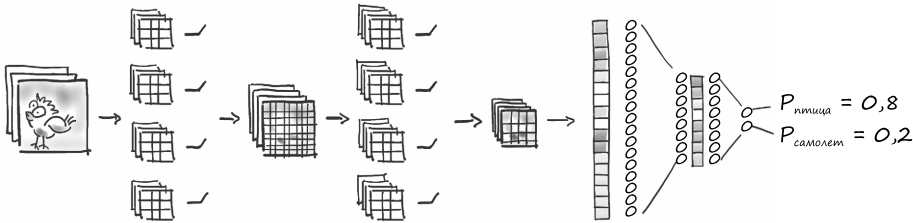


Рис. 8.9. Типовая схема сверточной сети, соответствующая в том числе и создаваемой нами тут. Изображение подается на вход ряда модулей свертки и субдискретизации с выбором максимального значения, после чего разворачивается в одномерный вектор и подается на вход полносвязных модулей

Забудем на минуту про комментарий «что-то пропущено». Обратим сначала внимание на то, что размер линейного слоя зависит от ожидаемого размера выходного сигнала слоя MaxPool2d: $8 \times 8 \times 8 = 512$. Посчитаем количество параметров этой небольшой модели:

```
# In[24]:
numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list

# Out[24]:
(18090, [432, 16, 1152, 8, 16384, 32, 64, 2])
```

Вполне приемлемо для ограниченного набора таких маленьких изображений. Для повышения разрешающих возможностей модели можно увеличить количество выходных каналов сверточных слоев (то есть число признаков, генерируемых каждым из сверточных слоев), в результате чего увеличится и размер линейного слоя.

Впрочем, примечание «Внимание» в коде было совсем не случайным. Модель обязательно пожалуется при запуске:

```
# In[25]:
model(img.unsqueeze(0))

# Out[25]:
...
RuntimeError: size mismatch, m1:
  [64 x 8], m2: [512 x 32] at c:\...\THTensorMath.cpp:940
```

Следует признать, это сообщение об ошибке выглядит довольно туманно, но не слишком. В трассировке можно заметить упоминания `linear`, а если взглянуть снова на модель, можно заметить, что единственный модуль с тензором 512×32 — это `nn.Linear(512, 32)`, первый линейный модуль, следующий за последним сверточным блоком.

Здесь недостает шага изменения формы, с 8-канального изображения 8×8 на состоящий из 512 элементов одномерный вектор (одномерный, если не считать измерения батчей). Это можно сделать, вызвав метод `view` для выходного сигнала `nn.MaxPool2d`, но, к сожалению, у нас нет явного доступа к выходным сигналам модулей при использовании `nn.Sequential`¹.

8.3. СОЗДАНИЕ ПОДКЛАССОВ `nn.Module`

На определенном этапе разработки нейронных сетей возникает необходимость вычислить что-то, не охваченное уже готовыми модулями. В нашем случае это такая простая вещь, как изменение формы²; но в подразделе 8.5.3 мы воспользуемся той же конструкцией для реализации остаточных связей. Так что в этом разделе мы научимся создавать свои собственные подклассы `nn.Module`, которые можно использовать точно так же, как уже готовые или как `nn.Sequential`.

Для создания моделей, способных на вещи более сложные, чем просто применение одного слоя за другим, необходимо использовать вместо `nn.Sequential` нечто с большей гибкостью. PyTorch позволяет производить в модели любые вычисления путем создания подклассов `nn.Module`.

Для создания подкласса `nn.Module` как минимум необходимо описать функцию `forward`, принимающую входные сигналы модуля и возвращающую выходной. Именно в ней и описываются производимые модулем вычисления. Название `forward` — это наследие давнего прошлого, когда в модуле должны были быть описаны как прямой, так и обратный проходы, встречавшиеся нам в подразделе 5.5.1. При использовании PyTorch и стандартных операций `torch` модуль `autograd` автоматически производит обратный проход; и действительно, `nn.Module` никогда не содержит `backward`.

Обычно в вычислениях используются и другие модули — готовые, например, свертки или пользовательские. Эти *подмодули* (*submodules*) обычно включаются в программу посредством описания в конструкторе `__init__` и присваивания

¹ Создатели PyTorch осознанно исключили возможность выполнения подобных операций изнутри `nn.Sequential` и надолго покинули этот путь; см. комментарии @soumith по адресу <https://github.com/pytorch/pytorch/issues/2486>. Недавно в PyTorch появился слой `nn.Flatten`.

² Начиная с PyTorch 1.3, можно воспользоваться `nn.Flatten` для этой цели.

их `self` для использования в функции `forward`. Их параметры в то же время хранятся в них на протяжении всего жизненного цикла нашего модуля. Обратите внимание, что перед этим необходимо вызвать `super().__init__()` (иначе PyTorch напомнит вам об этом).

8.3.1. Наша сеть как подкласс `nn.Module`

Напишем нашу сеть в виде подмодуля. Для этого создадим все экземпляры всех слоев `nn.Conv2d`, `nn.Linear` и т. д., которые выше передавали в конструктор, а затем воспользуемся этими экземплярами один за другим в функции `forward`:

```
# In[26]:
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.Tanh()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.Tanh()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.Tanh()
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8)
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

← Шаг изменения формы, который отсутствовал у нас ранее

Класс `Net` эквивалентен нашей предыдущей модели `nn.Sequential` в значении подмодулей, но благодаря написанию функции `forward` явным образом можно непосредственно производить операции над выходным сигналом `self.pool3` и вызвать для него `view`, чтобы преобразовать его в вектор $B \times N$. Обратите внимание, что в вызове `view` мы указываем `-1` для измерения батчей, поскольку в принципе не можем знать, сколько примеров данных будет содержать батч.

Здесь вся наша модель содержится в подклассе `nn.Module`. Можно также использовать подклассы для описания новых стандартных блоков в более сложных сетях. Если воспользоваться схемой в стиле главы 6, наша сеть будет выглядеть так, как показано на рис. 8.10. Мы выбираем по ситуации, какую информацию и где отражать.

Напомним, что задача классификационных сетей обычно заключается в сжатии информации в том смысле, что мы начинаем с изображения, содержащего

значительное количество пикселей, и сжимаем его в вектор вероятностей классов. С учетом этой цели стоит прокомментировать два связанных с этой архитектурой нюанса.

Во-первых, достижение нашей цели отражает сокращение объема промежуточных значений путем сокращения числа каналов в операциях свертки, сокращения количества пикселей посредством субдискретизации и благодаря тому, что размерность выходных данных меньше, чем размер входных данных в линейных сетях. Это общая особенность всех предназначенных для классификации сетей. Впрочем, во многих популярных архитектурах, например ResNet, которые мы встречали в главе 2 и обсудим подробнее в подразделе 8.5.3, подобное сокращение размера достигается за счет субдискретизации по пространственному разрешению, но количество каналов растет (и в целом размер все равно уменьшается). Похоже, что наш способ быстрого сокращения объемов информации неплохо работает для сетей ограниченной глубины и маленьких изображений; но для более глубоких сетей подобное сокращение обычно происходит медленнее.

Во-вторых, в одном из слоев, а именно при начальной свертке, выходной размер не сокращается по сравнению со входным. Если считать отдельный выходной пиксель вектором из 32 элементов (каналов), происходит линейное преобразование 27 элементов (свертка 3 каналов \times размер ядра 3×3), то есть происходит лишь незначительное увеличение размера. В ResNet в результате начальной свертки из 147 элементов (3 канала \times размер ядра 7×7) генерируется 64 канала¹. Таким образом, первый слой отличается от остальных тем, что сильно увеличивает объем в целом (в значении произведения числа каналов на количество пикселей) проходящих через него данных, но для каждого выходного пикселя по отдельности количество выходных сигналов все равно приблизительно совпадает с количеством входных.

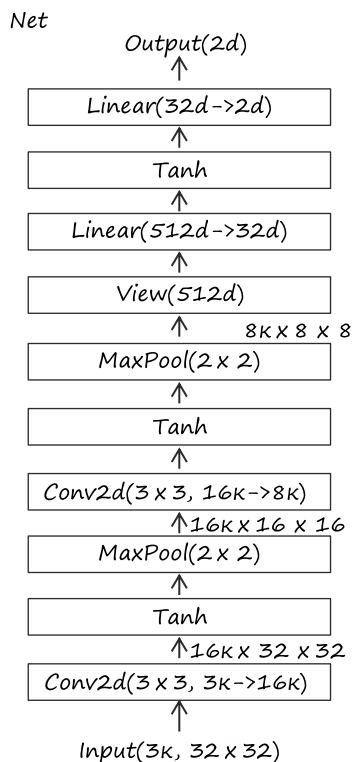


Рис. 8.10. Наша эталонная архитектура сверточной сети

¹ Вне сферы глубокого обучения (и до его появления) проекция в многомерное пространство с последующим принципиально более простым (по сравнению с линейным) машинным обучением известна под названием «ядерный трюк» (kernel trick). Начальный рост количества каналов можно считать явлением сходного порядка, но с другим соотношением изящества вложения и простоты работающей с ним модели.

8.3.2. Как PyTorch отслеживает параметры и подмодули

Любопытно, что присвоение экземпляра `nn.Module` атрибуту `nn.Module`, как было сделано в приведенном выше конструкторе, приводит к автоматической регистрации этого модуля в качестве подмодуля.

ПРИМЕЧАНИЕ

Подмодули должны быть атрибутами верхнего уровня, а не быть «закопаны» внутри экземпляров `list` или `dict`! В противном случае оптимизатор не сможет их (а значит, и их параметры) найти. На случай, если модели потребуется список или ассоциативный массив подмодулей, в PyTorch есть классы `nn.ModuleList` и `nn.ModuleDict`.

Можно вызывать любые методы подкласса `nn.Module`. Например, в модели, где процесс обучения существенно отличается от, скажем, предсказания, не помешает метод `predict`. Учтите, что вызовы подобных методов аналогичны вызову `forward` вместо самого модуля: они ничего не знают о точках привязки, и JIT при их использовании не видит структуры модуля, поскольку отсутствует эквивалент элементов `__call__`, показанных в подразделе 6.2.1.

Благодаря этому для доступа из `Net` к параметрам подмодулей не требуется каких-либо дополнительных действий пользователя:

```
# In[27]:
model = Net()

numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list

# Out[27]:
(18090, [432, 16, 1152, 8, 16384, 32, 64, 2])
```

Здесь вызов `model.parameters()` заходит во все подмодули, присвоенные атрибутам в конструкторе, и рекурсивно вызывает их методы `parameters()`. Вне зависимости от степени вложенности подмодуля, любой объект `nn.Module` может получить доступ к списку всех дочерних параметров. А обращаясь к их заполненному `autograd` атрибуту `grad`, оптимизатор знает, как модифицировать параметры так, чтобы снизить потери. Все это известно нам из главы 5.

Теперь мы знаем, как создавать свои собственные модули, и эти знания нам очень даже пригодятся в части II. Если вернуться к реализации класса `Net` и задуматься о целесообразности регистрации подмодулей в конструкторе для обращения к их параметрам, возникает впечатление, что регистрация подмодулей без параметров, например `nn.Tanh` и `nn.MaxPool2d`, — только лишняя трата ресурсов. Не проще ли было бы вызывать их непосредственно в функции `forward` точно так же, как мы вызывали `view`?

8.3.3. Функциональные API

Конечно, проще! Именно поэтому в PyTorch есть *функциональные* аналоги для всех модулей `nn`. Под функциональными мы подразумеваем «без внутреннего состояния» — другими словами, «выходное значение которых целиком и полностью определяется значениями входных аргументов». И действительно, `torch.nn.functional` предоставляет множество функций, работающих аналогично модулям из `nn`. Но вместо работы со входными аргументами и хранимыми параметрами, подобно аналогичным модулям, они принимают входные данные и параметры в качестве аргументов вызова функции. Например, функциональный аналог `nn.Linear` — `nn.functional.linear` — представляет собой функцию с сигнатурой `linear(input, weight, bias=None)`. Параметры `weight` и `bias` представляют собой аргументы функции.

Возвращаясь к нашей модели, имеет смысл продолжать пользоваться модулями `nn` для `nn.Linear` и `nn.Conv2d`, чтобы класс `Net` мог производить операции с их объектами `Parameter` во время обучения. Впрочем, можно спокойно перейти на функциональные аналоги субдискретизации и активации, поскольку у них параметров нет:

```
# In[28]:
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = out.view(-1, 8 * 8 * 8)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Намного лаконичнее и полностью эквивалентно предыдущему описанию класса `Net` из подраздела 8.3.1. Обратите внимание, что по-прежнему имеет смысл создать экземпляры модулей, требующих несколько параметров для их инициализации в конструкторе.

СОВЕТ

И хотя вы все еще можете найти универсальные научные функции, например `Tanh`, в версии 1.0 модуля `torch.nn.functional`, от этих точек входа постепенно отказываются в пользу функций из пространства имен `torch` верхнего уровня. Более узкоспециализированные функции, например `max_pool2d`, останутся в `torch.nn.functional`.

Таким образом, функциональный способ решения этой задачи также проливает свет на сущность API `nn.Module`: объект `Module` — это контейнер для состояния модели в виде объектов `Parameter` и подмодулей в сочетании с инструкциями по выполнению прямого прохода.

Что использовать — функциональный или модульный API, — зависит от вашего стиля и вкуса. Когда часть сети столь проста, что лучше воспользоваться `nn.Sequential`, — мы в царстве модулей. При написании же своих собственных функций `forward` логичнее будет воспользоваться функциональным интерфейсом для тех частей, которым не требуется состояние в виде параметров.

В главе 15 мы вкратце затронем вопрос квантования. При этом части сети без сохранения состояния, например функции активации, внезапно начнут требовать состояния, поскольку нужно будет захватывать информацию о квантовании. Это значит, что, если вы намерены квантовать модель, имеет смысл остановиться на модульном API, если речь идет о не-JIT-квантовании. Одно стилистическое соображение, которое поможет избежать сюрпризов в случае непредвиденных сценариев использования: при потребности в различных приложениях модулей без сохранения состояния (например, `nn.HardTanh` или `nn.ReLU`), вероятно, имеет смысл предусмотреть для каждого отдельный экземпляр. Переиспользование одних и тех же модулей может показаться более изящным решением, дающим в данный момент правильные результаты при стандартном применении Python, но утилиты для анализа модели могут на этом «споткнуться».

Итак, теперь мы можем при необходимости создавать свои собственные объекты `nn.Module`, а на случай, когда создание экземпляра и вызов `nn.Module` — перебор, у нас есть функциональный API. Это был последний фрагмент, необходимый для понимания организации кода практически во всех нейронных сетях, реализуемых с помощью PyTorch.

Давайте еще раз проверим, что наша модель работает, а затем перейдем к циклу обучения:

```
# In[29]:
model = Net()
model(img.unsqueeze(0))

# Out[29]:
tensor([[ -0.0157,  0.1143]], grad_fn=<AddmmBackward>)
```

Получили два числа! Информация проходит через модель должным образом. Возможно, вы до этого момента не осознавали, что добиться правильного размера первого линейного слоя в более сложных моделях иногда непросто. Мы слышали истории о знаменитых специалистах, бравших произвольные числа, а затем производивших трассировку сообщений об ошибке PyTorch для получения правильного размера линейных слоев. Довольно неуклюже, правда? Но нет, все абсолютно нормально!

8.4. ОБУЧАЕМ НАШУ СВЕРТОЧНУЮ СЕТЬ

Мы достигли этапа, на котором можем собрать полный цикл обучения воедино. Мы уже разработали его общую структуру в главе 5, и цикл обучения очень напоминает цикл обучения из главы 6, но осталось добавить в него некоторые детали, например отслеживание степени безошибочности. После запуска модели мы также захотим немного ускорить ее работу, так что научимся ускорять работу моделей с помощью GPU. Но сначала взглянем на цикл обучения.

Напоминаем, что в основе нашей сверточной сети лежат два вложенных цикла: внешний — по *эпохам*, а внутренний — на основе объекта `DataLoader`, генерирующего батчи из объекта `Dataset`. На каждой итерации цикла необходимо сделать следующее.

1. Пропустить входные сигналы через модель (прямой проход).
2. Вычислить функцию потерь (также часть прямого прохода).
3. Обнулить все старые градиенты.
4. Вызвать `loss.backward()` для вычисления градиентов функции потерь относительно каждого из параметров (обратный проход).
5. Оптимизировать в сторону уменьшения потерь.

Кроме того, мы собираем и выводим на экран определенную информацию. Итак, вот наш цикл обучения: практически такой же, как и в предыдущей главе, но все равно напомним, что делает каждая его часть:

```
# In[30]:
import datetime
```

Используем встроенный модуль работы с временем и датой PyTorch

```
def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
```

Цикл по эпохам, пронумерованным от 1 до `n_epochs` и не начинающимся с 0

```
    for epoch in range(1, n_epochs + 1):
```

Пропускаем батч через нашу модель...

```
        loss_train = 0.0
        for imgs, labels in train_loader:
```

Проходим в цикле по нашему набору данных по батчам, создаваемым загрузчиком данных

```
            outputs = model(imgs)
```

...и вычисляем минимизируемую функцию потерь

```
            loss = loss_fn(outputs, labels)
```

Избавившись от градиентов с предыдущей итерации...

```
            optimizer.zero_grad()
```

...выполняем обратный проход. То есть вычисляем градиенты по всем обучаемым параметрам сети

```
            loss.backward()
```

Обновляем модель

```
            optimizer.step()
```

Суммируем потери за эпоху. Напомним, что важно преобразовать значение потерь в числовое значение Python с помощью `.item()` для экранирования градиентов

```
            loss_train += loss.item()
```

```

if epoch == 1 or epoch % 10 == 0:
    print('{} Epoch {}, Training loss {}'.format(
        datetime.datetime.now(), epoch,
        loss_train / len(train_loader)))

```

Делим на длину загрузчика обучающих данных для получения средних потерь на батч — намного более интуитивная мера, чем сумма

Мы берем объект `Dataset` из главы 7, оборачиваем его в `DataLoader`, создаем экземпляр класса сети, оптимизатор и функцию потерь, как и раньше, и затем вызываем цикл обучения.

Существенные изменения в сравнении с нашей моделью из предыдущей главы: теперь модель представляет собой пользовательский подкласс `nn.Module`, и мы используем свертки. Запустим обучение в течение 100 эпох, выводя по ходу процесса значения потерь. В зависимости от вашего аппаратного обеспечения, выполнение может занять 20 минут или даже больше!

```

# In[31]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = Net() #
optimizer = optim.SGD(model.parameters(), lr=1e-2) #
loss_fn = nn.CrossEntropyLoss() #

training_loop(
    n_epochs = 100,
    optimizer = optimizer,
    model = model,
    loss_fn = loss_fn,
    train_loader = train_loader,
)

# Out[31]:
2020-01-16 23:07:21.889707 Epoch 1, Training loss 0.5634813266954605
2020-01-16 23:07:37.560610 Epoch 10, Training loss 0.3277610331109375
2020-01-16 23:07:54.966180 Epoch 20, Training loss 0.3035225479086493
2020-01-16 23:08:12.361597 Epoch 30, Training loss 0.28249378549824855
2020-01-16 23:08:29.769820 Epoch 40, Training loss 0.2611226033253275
2020-01-16 23:08:47.185401 Epoch 50, Training loss 0.24105800626574048
2020-01-16 23:09:04.644522 Epoch 60, Training loss 0.21997178820477928
2020-01-16 23:09:22.079625 Epoch 70, Training loss 0.20370126601047578
2020-01-16 23:09:39.593780 Epoch 80, Training loss 0.18939699422401987
2020-01-16 23:09:57.111441 Epoch 90, Training loss 0.17283396527266046
2020-01-16 23:10:14.632351 Epoch 100, Training loss 0.1614033816868712

```

Объект `DataLoader` организует примеры данных из нашего набора данных `cifar2` по батчам. Перетасовка обеспечивает случайный порядок примеров данных из набора

...и оптимизатор на основе стохастического градиентного спуска, с которым мы работали...

Создаем экземпляр класса сети...

...и функцию потерь на основе перекрестной энтропии, знакомую нам из раздела 7.10

Вызываем описанный выше цикл обучения

Итак, мы уже можем обучить нашу сеть. Но опять же наш друг — любительница птиц, вероятно, будет не слишком впечатлена, если услышит, что мы обучили сеть до очень низких значений потерь на обучающем наборе данных.

8.4.1. Измерение степени безошибочности

В качестве более вразумительной, по сравнению с функцией потерь, меры можно рассмотреть показатели безошибочности на обучающем и проверочном наборах данных. Воспользуемся кодом из главы 7:

```
# In[32]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=False)
val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=64,
                                         shuffle=False)

def validate(model, train_loader, val_loader):
    for name, loader in [("train", train_loader), ("val", val_loader)]:
        correct = 0
        total = 0
        with torch.no_grad():
            for imgs, labels in loader:
                outputs = model(imgs)
                _, predicted = torch.max(outputs, dim=1)
                total += labels.shape[0]
                correct += int((predicted == labels).sum())
            print("Accuracy {}: {:.2f}".format(name, correct / total))

validate(model, train_loader, val_loader)
```

Градиенты нам здесь не нужны, так как мы не собираемся обновлять значения параметров

Получаем индекс максимального значения

Подсчитываем количество примеров данных, поэтому total увеличивается на размер батча

Сравнивая предсказанный класс, обладающий максимальной вероятностью, с эталонными метками, получаем сначала булев массив. Его суммирование дает нам количество элементов батча, для которых предсказание и эталонное значение совпадают

```
# Out[32]:
Accuracy train: 0.93
Accuracy val: 0.89
```

Приводим к типу `int` языка Python, для целочисленных тензоров это эквивалентно использованию метода `.item()`, аналогично тому, как мы делали в цикле обучения.

Результаты получились намного лучше, чем у полносвязной модели, достигавшей безошибочности лишь в 79 %. Мы почти вдвое сократили количество ошибочных предсказаний на проверочном наборе данных. Кроме того, мы обошлись намного меньшим числом параметров. А значит, эта модель лучше обобщает на новый пример данных задачу распознавания содержания изображений, посредством локальности и инвариантности относительно сдвига. Можно теперь запустить ее в течение намного большего числа эпох и посмотреть, каких результатов удастся добиться.

8.4.2. Сохранение и загрузка модели

Поскольку мы пока что удовлетворены результатами работы модели, не мешает ее сохранить, правда? Сделать это очень просто. Давайте сохраним модель в файл:

```
# In[33]:
torch.save(model.state_dict(), data_path + 'birds_vs_airplanes.pt')
```

Файл `birds_vs_airplanes.pt` теперь содержит все параметры объекта `model`: весовые коэффициенты и смещения для двух модулей свертки и двух линейных модулей. Да, никакой структуры, только весовые коэффициенты. Это значит, что при развертывании модели для нашего друга в реальных условиях нам понадобится описание класса `model`, а еще нужно будет создать экземпляр и затем загрузить в него обратно параметры:

```
# In[34]:
loaded_model = Net()
loaded_model.load_state_dict(torch.load(data_path
                                       + 'birds_vs_airplanes.pt'))
```

Нужно убедиться, что мы не меняем определение сети между сохранением и последующей загрузкой состояния модели

```
# Out[34]:
<All keys matched successfully>
```

Мы также включили соответствующую предобученную модель в наш репозиторий кода, в файл `../data/p1ch7/birds_vs_airplanes.pt`.

8.4.3. Обучение на GPU

У нас есть сеть, и мы умеем ее обучать! Но нам все же не мешает делать это несколько быстрее. Так что вы вряд ли удивитесь, что теперь мы перенесем процесс обучения на GPU. С помощью метода `.to`, уже встречавшегося нам в главе 3, можно перенести полученные от загрузчика данных тензоры в GPU, после чего обучение автоматически будет производиться там. К счастью, в `nn.Module` реализована функция `.to`, перемещающая все параметры в GPU (или приводящая тип данных, если передать ей аргумент `dtype`).

Между `Module.to` и `Tensor.to` существует тонкое различие. `Module.to` производит операции с заменой на месте, то есть изменяет экземпляр модуля. А `Tensor.to` — нет (в некотором смысле аналогично `Tensor.tanh`), возвращая новый тензор. Одно из следствий этого: рекомендуемой практикой является создание экземпляра `Optimizer` после перемещения всех параметров на нужное устройство.

Перенос вычислений на GPU при его наличии считается хорошим стилем программирования. Неплохим паттерном программирования будет установка значения переменной `device` в зависимости от `torch.cuda.is_available`:

```
# In[35]:
device = (torch.device('cuda') if torch.cuda.is_available()
          else torch.device('cpu'))
print(f"Training on device {device}.")
```

Далее можно внести соответствующие изменения в цикл обучения, переместив полученные от загрузчика данных тензоры на GPU с помощью метода `Tensor.to`. Обратите внимание, что код в точности соответствует первой его версии в начале этого раздела, за исключением двух строк переноса входных данных на GPU:

```
# In[36]:
import datetime

def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            imgs = imgs.to(device=device)
            labels = labels.to(device=device)
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            loss_train += loss.item()

        if epoch == 1 or epoch % 10 == 0:
            print('{ } Epoch { }, Training loss { }'.format(
                datetime.datetime.now(), epoch,
                loss_train / len(train_loader)))
```

← Эти две строки для переноса `imgs` и `labels` на устройство, на котором будет производиться обучение, — единственное отличие от предыдущей версии

Те же самые изменения необходимо внести в функцию `validate`, после чего можно создать экземпляр модели, перенести ее на `device` и запустить так же, как раньше¹:

```
# In[37]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = Net().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-2)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs = 100,
    optimizer = optimizer,
    model = model,
```

← Переносит нашу модель (все ее параметры) на GPU. Если забыть перенести саму модель или входные данные на GPU, вы получите сообщения об ошибках, указывающие, что тензоры располагаются на различных устройствах, поскольку операторы PyTorch не поддерживают смеси входных данных на GPU и CPU

¹ У загрузчика данных есть опция `pin_memory`, при указании которой он использует закрепленную память GPU, с целью ускорения переноса данных. Насколько это помогает, впрочем, зависит от многих условий, так что мы не станем развивать здесь эту тему.

```

    loss_fn = loss_fn,
    train_loader = train_loader,
)

# Out[37]:
2020-01-16 23:10:35.563216 Epoch 1, Training loss 0.5717791349265227
2020-01-16 23:10:39.730262 Epoch 10, Training loss 0.3285350770137872
2020-01-16 23:10:45.906321 Epoch 20, Training loss 0.29493294959994637
2020-01-16 23:10:52.086905 Epoch 30, Training loss 0.26962305994550134
2020-01-16 23:10:56.551582 Epoch 40, Training loss 0.24709946277794564
2020-01-16 23:11:00.991432 Epoch 50, Training loss 0.22623272664892446
2020-01-16 23:11:05.421524 Epoch 60, Training loss 0.20996672821462534
2020-01-16 23:11:09.951312 Epoch 70, Training loss 0.1934866009719053
2020-01-16 23:11:14.499484 Epoch 80, Training loss 0.1799132404908253
2020-01-16 23:11:19.047609 Epoch 90, Training loss 0.16620008706761774
2020-01-16 23:11:23.590435 Epoch 100, Training loss 0.15667157247662544

```

Даже в случае нашей маленькой сети ускорение работы довольно значительное. Преимущества вычислений на GPU более заметны на крупных моделях.

Небольшая сложность при загрузке весовых коэффициентов сети: PyTorch попытается загрузить веса на то же устройство, с которого они были сохранены, то есть весовые коэффициенты с GPU будут восстановлены на GPU. Поскольку неизвестно, нужно ли нам то же устройство, существует два варианта: перенести сеть на CPU перед сохранением или вернуть ее обратно после восстановления. Более лаконичным вариантом будет потребовать от PyTorch переопределить информацию об устройстве при загрузке весовых коэффициентов. Сделать это можно посредством передачи методу `torch.load` ключевого аргумента `map_location`:

```

# In[39]:
loaded_model = Net().to(device=device)
loaded_model.load_state_dict(torch.load(data_path
                                       + 'birds_vs_airplanes.pt',
                                       map_location=device))

# Out[39]:
<All keys matched successfully>

```

8.5. АРХИТЕКТУРА МОДЕЛИ

Мы создали модель в виде подкласса `nn.Module` — фактического стандарта для всех моделей, кроме простейших. Затем мы успешно обучили ее и научились использовать GPU для обучения наших моделей. Мы достигли этапа, когда можем создать сверточную нейронную сеть прямого распространения и успешно обучить ее классифицировать изображения. Возникает естественный вопрос: что теперь? Что, если нам поставят более сложную задачу? Надо признать, наш набор данных с птицами и самолетами не такой уж и сложный: изображения

были очень маленькими, а изучаемые объекты располагались в них по центру и занимали большую часть поля зрения.

Если обратиться, скажем, к ImageNet, мы столкнемся с более крупными и сложными изображениями, в которых правильный ответ зависит от множества визуальных зацепок, зачастую организованных иерархически. Например, чтобы предсказать, является ли темный объект прямоугольной формы пультом дистанционного управления или мобильным телефоном, сеть могла бы искать нечто напоминающее экран.

Кроме того, в реальном мире изображения не единственная наша цель, ведь существуют еще табличные данные, последовательности и текст. Нейронные сети одновременно обещают достаточную гибкость для решения задач для всех этих видов данных при должной архитектуре (то есть комбинации слоев или модулей) и должной функции потерь.

PyTorch включает всеобъемлющий набор модулей и функций потерь для реализации самых современных архитектур, от компонентов прямого пространства до модулей с долгой краткосрочной памятью (long short-term memory, LSTM) и сетей-преобразователей (две очень популярные архитектуры для обработки последовательных данных). В PyTorch Hub, а также в качестве части torchvision и других инициатив сообщества разработчиков доступно немало моделей.

Мы рассмотрим несколько более развитых архитектур в части II, где пошагово пройдем по комплексной задаче анализа КТ-снимков, но в целом обсуждение различных вариантов архитектур нейронных сетей выходит за рамки темы данной книги. Впрочем, на основе уже накопленных нами знаний можно разобаться, как создать практически любую архитектуру благодаря выразительности PyTorch. Цель этого раздела как раз и заключается в обеспечении степени понимания, достаточной, чтобы прочитать свежие научные работы и приступить к реализации их на PyTorch, либо — поскольку авторы зачастую публикуют реализации программ из своих статей на PyTorch — спокойно читать эти реализации за чашечкой кофе.

8.5.1. Расширение объема памяти: ширина

С учетом используемой нами архитектуры прямого распространения есть несколько вопросов, которые мы хотели бы обсудить подробнее, прежде чем углубиться в более сложные нюансы. Первый из этих вопросов — *ширина* сети: количество нейронов в слое или каналов на каждую операцию свертки. Расширить модель в PyTorch очень легко. Необходимо просто указать большее количество выходных каналов в первой свертке и увеличивать следующие слои соответствующим образом, не забывая менять функцию forward так,

чтобы отразить увеличившуюся длину вектора при переходе на полносвязные слои:

```
# In[40]:
class NetWidth(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 16, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(16 * 8 * 8, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = out.view(-1, 16 * 8 * 8)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Чтобы не нужно было жестко «зашивать» числа в описании модели, удобно передавать параметр `init` и параметризовать ширину модели, не забывая также добавить параметр в вызов `view` в функции `forward`:

```
# In[42]:
class NetWidth(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3,
                                padding=1)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Количество каналов и признаков для каждого слоя непосредственно связано с числом параметров модели: при прочих равных условиях они повышают *разрешающие возможности* (*capacity*) модели. Можно посмотреть, как мы уже делали ранее, сколько параметров у нашей модели теперь:

```
# In[44]:
sum(p.numel() for p in model.parameters())

# Out[44]:
38386
```

Чем больше разрешающие возможности модели, тем с большей степенью изменчивости входных сигналов сможет справиться модель, но в то же время тем выше вероятность переобучения, поскольку модель сможет воспользоваться дополнительными параметрами для запоминания несущественных аспектов входных данных. Мы уже обсуждали способы борьбы с переобучением, лучший из которых — увеличение размера выборки, или, в отсутствие новых данных, дополнение данных посредством искусственных изменений уже существующих.

Есть еще несколько приемов на уровне модели (без изменения данных) для борьбы с переобучением. Давайте рассмотрим наиболее распространенные из них.

8.5.2. Улучшаем сходимость модели и ее способности к обобщению: регуляризация

Обучение модели включает два важнейших шага: оптимизацию, при которой мы стремимся к уменьшению функции потерь на обучающем наборе данных, и обобщение, когда модели приходится работать не только на обучающем наборе данных, но также и на данных, которые она ранее не встречала, например на проверочном наборе данных. Математические инструменты упрощения этих двух шагов обычно объединяют под термином «регуляризация» (*regularization*).

Держим параметры под контролем: штрафы на весовые коэффициенты

Первый способ достижения устойчивости обобщения: добавление члена регуляризации в формулу потерь. Этот дополнительный член ограничивает рост весовых коэффициентов модели в процессе обучения: он устроен так, что они стремятся оставаться маленькими. Другими словами, он налагает штраф на большие значения весов. В результате форма функции потерь становится более гладкой, и для модели нет особого смысла подстраиваться под отдельные примеры данных.

Наиболее популярные виды членов регуляризации: L2-регуляризация (сумма квадратов всех весовых коэффициентов модели) и L1-регуляризация (сумма абсолютных значений всех весовых коэффициентов модели)¹. Оба они масштабируются на (малый) коэффициент: задаваемый до обучения гиперпараметр.

¹ Мы сосредоточим свое внимание на L2-регуляризации. L1-регуляризация — известная из более общей литературы по статистике благодаря использованию в Lasso — обладает такой привлекательной характеристикой, как итоговые разреженные весовые коэффициенты после обучения.

L2-регуляризацию также называют *затуханием весов* (*weight decay*). Дело в том, что в SGD и обратном распространении ошибки отрицательный градиент L2-регуляризации по параметру w_i равен $2 * \lambda * w_i$, где λ — вышеупомянутый гиперпараметр, который просто называется в PyTorch *затуханием веса* (*weight decay*). Поэтому прибавление к функции потерь члена L2-регуляризации эквивалентно уменьшению каждого весового коэффициента пропорционально его текущему значению во время шага оптимизации (отсюда и название «затухание веса»). Обратите внимание, что затухание веса относится ко всем параметрам сети, в том числе и к смещениям.

В PyTorch можно довольно легко реализовать регуляризацию путем добавления коэффициента в формулу функции потерь. После вычисления функции потерь, какая бы она ни была, можно пройти в цикле по параметрам модели, суммируя их квадраты (для L2) или `abs` (для L1), и произвести обратное распространение ошибки:

```
# In[45]:
def training_loop_l2reg(n_epochs, optimizer, model, loss_fn,
                       train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            imgs = imgs.to(device=device)
            labels = labels.to(device=device)
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            l2_lambda = 0.001
            l2_norm = sum(p.pow(2.0).sum()
                           for p in model.parameters())
            loss = loss + l2_lambda * l2_norm

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        loss_train += loss.item()
    if epoch == 1 or epoch % 10 == 0:
        print('{} Epoch {}, Training loss {}'.format(
            datetime.datetime.now(), epoch,
            loss_train / len(train_loader)))
```

Для L1-регуляризации —
замените `pow(2.0)` на `abs()`

Впрочем, в оптимизаторе SGD в PyTorch уже есть параметр `weight_decay`, соответствующий $2 * \lambda$, который напрямую осуществляет затухание весов во время их обновления, как описывалось выше. Он полностью эквивалентен прибавлению L2-нормы весовых коэффициентов к функции потерь без необходимости накопления в функции потерь и вовлечения автоматического вычисления градиентов.

Не слишком полагаемся на отдельные входные сигналы: дропаут

В так и названной — «Дропаут: простой способ предотвратить переобучение нейронных сетей» (Dropout: a Simple Way to Prevent Neural Networks from Overfitting, <http://mng.bz/nPMa>) — статье 2014 года Нитиша Шриваставы (Nitish Srivastava) с соавторами из исследовательской группы Джефа Хинтона (Geoff Hinton) из Торонто была впервые представлена эффективная стратегия борьбы с переобучением. Судя по названию, это как раз то, что нам нужно, не так ли? Идея дропаута действительно проста: обнуляем случайную часть выходных сигналов нейронов по сети, причем этот случайный выбор производится на каждой итерации обучения.

Фактически в результате этой процедуры на каждой итерации формируются слегка отличающиеся модели с различными топологиями нейронов, уменьшая шансы нейронов модели скоординироваться в процессе запоминания, что происходит при переобучении. Можно также считать, что дропаут вносит возмущения в генерируемые моделью признаки, производя эффект, схожий с дополнением данных, но на этот раз по всей сети.

В PyTorch можно реализовать дропаут в модели с помощью добавления модуля `nn.Dropout` между нелинейной функцией активации и линейным или сверточным модулем последующего слоя. В качестве аргумента необходимо указать вероятность, с которой будут обнуляться входные сигналы. Для сверток необходимо использовать специализированные слои `nn.Dropout2d` или `nn.Dropout3d`, обнуляющие целые каналы входных сигналов:

```
# In[47]:
class NetDropout(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv1_dropout = nn.Dropout2d(p=0.4)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3,
                               padding=1)
        self.conv2_dropout = nn.Dropout2d(p=0.4)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = self.conv1_dropout(out)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = self.conv2_dropout(out)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```


Обратите внимание, что дропаут обычно происходит во время обучения, в то время как во время использования обученной модели в реальных условиях модуль дропаута обходят или, что эквивалентно, присваивают равную нулю вероятность. Этот процесс контролируется свойством `train` модуля `Dropout`. Напомним, что PyTorch позволяет переключаться между двумя режимами, вызывая

```
model.train()
```

или

```
model.eval()
```

для любого подкласса `nn.Model`. Вызов автоматически дублируется для всех подмодулей, так что, если среди них есть `Dropout`, он будет вести себя соответствующим образом в последующих прямых и обратных проходах.

Держим активацию под контролем: нормализация по батчам

Дропаут как раз был последним криком моды в 2015-м, когда Сергей Йоффе (Sergey Ioffe) и Кристиан Сегеди (Christian Szegedy) опубликовали еще одну статью по результатам семинара под названием «Нормализация по батчам: ускорение обучения нейронных сетей путем сокращения внутреннего ковариантного сдвига» (Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, <https://arxiv.org/abs/1502.03167>). В этой статье описывалась методика, оказывавшая положительное влияние на обучение с нескольких точек зрения, позволяя повысить скорость обучения и снизить зависимость обучения от начальных значений, а также игравшая роль регуляризатора, тем самым представляя альтернативное дропауту решение.

Основная идея нормализации по батчам состоит в нормализации входных сигналов функций активации сети так, чтобы получить определенное желательное распределение для мини-батчей. Если вспомнить внутренние механизмы обучения и роль нелинейных функций активации, становится ясно, что это помогает избежать чрезмерного углубления входных сигналов функций активации в область насыщения, что гасит градиенты и замедляет обучение.

На практике нормализация по батчам сдвигает и масштабирует промежуточные входные сигналы на основе среднего значения и стандартного отклонения, вычисленных в этой промежуточной точке по примерам данных мини-батча. Эффект от регуляризации основан на том, что отдельные примеры данных и следующие далее по конвейеру функции активации всегда рассматриваются моделью как сдвинутые и нормализованные, в зависимости от статистических показателей выделенного случайным образом мини-батча. Что само по себе является разновидностью *систематического* (*principled*) дополнения данных. Авторы статьи высказывают мнение, что нормализация по батчам исключает или по крайней мере сокращает необходимость в дропауте.

Нормализация по батчам в PyTorch производится с помощью модулей `nn.BatchNorm1D`, `nn.BatchNorm2d` и `nn.BatchNorm3d`, в зависимости от размерности входных данных. А поскольку цель нормализации по батчам — масштабировать входные сигналы функций активации, логично будет производить ее после линейного преобразования (свертки в данном случае) и активации, как показано здесь:

```
# In[49]:
class NetBatchNorm(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv1_batchnorm = nn.BatchNorm2d(num_features=n_chans1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3,
                                padding=1)
        self.conv2_batchnorm = nn.BatchNorm2d(num_features=n_chans1 // 2)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = self.conv1_batchnorm(self.conv1(x))
        out = F.max_pool2d(torch.tanh(out), 2)
        out = self.conv2_batchnorm(self.conv2(out))
        out = F.max_pool2d(torch.tanh(out), 2)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Как и дропаут, нормализация по батчам должна вести себя по-разному во время обучения и во время выполнения вывода. На самом деле во время выполнения вывода желательно, чтобы выходной сигнал для конкретного входного сигнала не зависел от прочих входных сигналов, подаваемых на вход модели. А это значит, что необходим способ нормализовать данные, но при этом раз и навсегда зафиксировать параметры нормализации.

При обработке мини-батчей, помимо оценки среднего значения и стандартного отклонения для текущего мини-батча, PyTorch также обновляет в качестве приближения скользящие оценки среднего значения и стандартного отклонения, репрезентативные для всего набора данных. Таким образом, если пользователь указывает

```
model.eval()
```

и модель содержит модуль нормализации по батчам, скользящие оценки фиксируются и используются для нормализации. Для разблокировки скользящих оценок и возврата к использованию статистических показателей мини-батчей мы вызываем `model.train()` точно так же, как делали для дропаута.

8.5.3. Забираемся глубже для усвоения более сложных структур: глубина сети

Ранее мы говорили о ширине как первом по порядку вопросе, который следует учесть, чтобы сделать модель больше и в каком-то смысле обладающей большими разрешающими возможностями. Второй основной вопрос, разумеется, *глубина*. А поскольку наша книга посвящена глубокому обучению, логично предположить, что глубина должна нас интересовать. В конце концов, более глубокие модели всегда лучше менее глубоких, правда? Ну, зависит от обстоятельств. С ростом глубины сложность функций, которые способна аппроксимировать сеть, в целом растет. Применительно к машинному зрению менее глубокая сеть способна распознать контуры человека на фотографии, а более глубокая — распознать человека, лицо в верхней его половине и рот на лице. Глубина обеспечивает возможность работы сети с иерархической информацией, когда для анализа какого-либо входного сигнала необходимо понимать контекст.

На глубину можно смотреть и с другой точки зрения: увеличение глубины связано с увеличением длины последовательности операций, выполняемых сетью при обработке входных данных. Такая точка зрения — глубокая сеть, решающая поставленную задачу посредством выполнения последовательности операций, — очень привлекательна для разработчиков программного обеспечения, привыкших думать об алгоритмах как последовательности операций наподобие «найти очертания человека, искать голову в их верхней части, искать рот внутри контура головы».

Обходные связи

Повышение глубины сети означает дополнительные сложности, из-за которых модели глубокого обучения достигли глубины в 20 и более слоев только в конце 2015 года. Углубление сети обычно ухудшает сходимость. Давайте вернемся к понятию обратного распространения ошибки в контексте очень глубоких сетей. Производные функций потерь относительно параметров, особенно в первых слоях, приходится умножать на множество других чисел, генерируемых цепочкой операций между функцией потерь и параметром. Эти множители могут быть маленькими, приводя в результате к еще меньшим числам, или большими, поглощая маленькие числа из-за приближенности операций с плавающей запятой. В сухом остатке мы получаем, что в результате длинной цепочки операций умножения вклад отдельного параметра в градиент *исчезает*, и это ведет к неэффективному обучению данного слоя, поскольку ни этот, ни другие параметры не будут обновляться должным образом.

В декабре 2015 года Каймин Хе (Kaiming He) с соавторами представили широкой общественности *остаточные сети* (*residual networks*, *ResNets*) — архитектуру, в которой для успешного обучения очень глубоких сетей применяется довольно простая уловка (<https://arxiv.org/abs/1512.03385>). Эта работа сделала возможными


```

self.fc1 = nn.Linear(4 * 4 * n_chans1 // 2, 32)
self.fc2 = nn.Linear(32, 2)

def forward(self, x):
    out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
    out = F.max_pool2d(torch.relu(self.conv2(out)), 2)
    out = F.max_pool2d(torch.relu(self.conv3(out)), 2)
    out = out.view(-1, 4 * 4 * self.n_chans1 // 2)
    out = torch.relu(self.fc1(out))
    out = self.fc2(out)
    return out

```

Добавление в эту модель обходной связи наподобие ResNet сводится к прибавлению выходного сигнала первого слоя в функции `forward` к входному сигналу третьего слоя:

```

# In[53]:
class NetRes(nn.Module):
    def __init__(self, n_chans1=32):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3,
                                padding=1)
        self.conv3 = nn.Conv2d(n_chans1 // 2, n_chans1 // 2,
                                kernel_size=3, padding=1)
        self.fc1 = nn.Linear(4 * 4 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        out = F.max_pool2d(torch.relu(self.conv2(out)), 2)
        out1 = out
        out = F.max_pool2d(torch.relu(self.conv3(out)) + out1, 2)
        out = out.view(-1, 4 * 4 * self.n_chans1 // 2)
        out = torch.relu(self.fc1(out))
        out = self.fc2(out)
        return out

```

Другими словами, мы воспользовались выходным сигналом первых функций активации в качестве входных сигналов для последних, помимо стандартного пути прямого распространения. Другое название этого процесса — *тождественное отображение* (*identity mapping*). Как же оно помогает решить проблему исчезающих градиентов, упомянутую нами ранее?

Если говорить об обратном распространении ошибки, удобно, что обходная связь (или последовательность обходных связей в глубокой сети) создает прямой путь от расположенных глубоко параметров к функции потерь, благодаря чему они вносят более непосредственный вклад в градиент функции потерь, ведь частные производные функции потерь по этим параметрам теперь получают шанс не умножаться на коэффициенты в длинной цепочке прочих операций.

Отмечается, что обходные связи благотворно влияют на сходимость, особенно на начальных этапах обучения. Кроме того, поверхность функции потерь глубоких остаточных сетей намного глаже, чем у сетей прямого распространения той же глубины и ширины.

Стоит отметить, что обходные связи не были чем-то новым на момент появления ResNet. Обходные связи уже использовались, в том или ином виде, в магистральных сетях (highway networks) и U-Net. Впрочем, способ применения обходных связей в ResNet дал возможность успешно обучать модели глубиной более 100 слоев.

С момента появления ResNet обходные связи вышли в других архитектурах сетей на новый уровень. В частности, в одной из них, DenseNet, предлагается связывать каждый слой с несколькими другими, расположенными далее в сети слоями посредством обходных связей, достигая результатов на самом передовом уровне при меньшем количестве параметров. И мы уже знаем, как можно реализовать что-то наподобие DenseNet: просто арифметически прибавить предыдущие промежуточные выходные сигналы к последующим промежуточным входным сигналам.

Создание очень глубоких моделей в PyTorch

Мы говорили о сверточных нейронных сетях более чем из 100 слоев. Как создать подобную сеть в PyTorch и не сойти с ума в процессе? Обычная стратегия: описать стандартный блок, например (Conv2d, ReLU, Conv2d) + блок обходной связи, а затем динамически создавать сеть в цикле `for`. Взглянем, как это происходит на практике. Мы создадим сеть, приведенную на рис. 8.12.

Сначала мы создадим подкласс модуля, единственная задача которого будет состоять в организации вычислений одной *блока*: одной группы операций свертки, функции активации и обходной связи:

```
# In[55]:
class ResBlock(nn.Module):
    def __init__(self, n_chans):
        super(ResBlock, self).__init__()
        self.conv = nn.Conv2d(n_chans, n_chans, kernel_size=3,
                               padding=1, bias=False)
        self.batch_norm = nn.BatchNorm2d(num_features=n_chans)
        torch.nn.init.kaiming_normal_(self.conv.weight,
                                       nonlinearity='relu')
        torch.nn.init.constant_(self.batch_norm.weight, 0.5)
        torch.nn.init.zeros_(self.batch_norm.bias)

    def forward(self, x):
        out = self.conv(x)
        out = self.batch_norm(out)
        out = torch.relu(out)
        return out + x
```

Слой BatchNorm свел бы на нет эффект смещения, так что его обычно опускают

Использует пользовательские функции активации. Начальные значения `kaiming_normal_` — случайные элементы из нормального распределения со стандартным отклонением, вычисленным в соответствии со статьей о ResNet. Начальные значения нормализации по батчам задаются таким образом, чтобы получить выходные распределения с начальным средним значением 0 и дисперсией 0,5

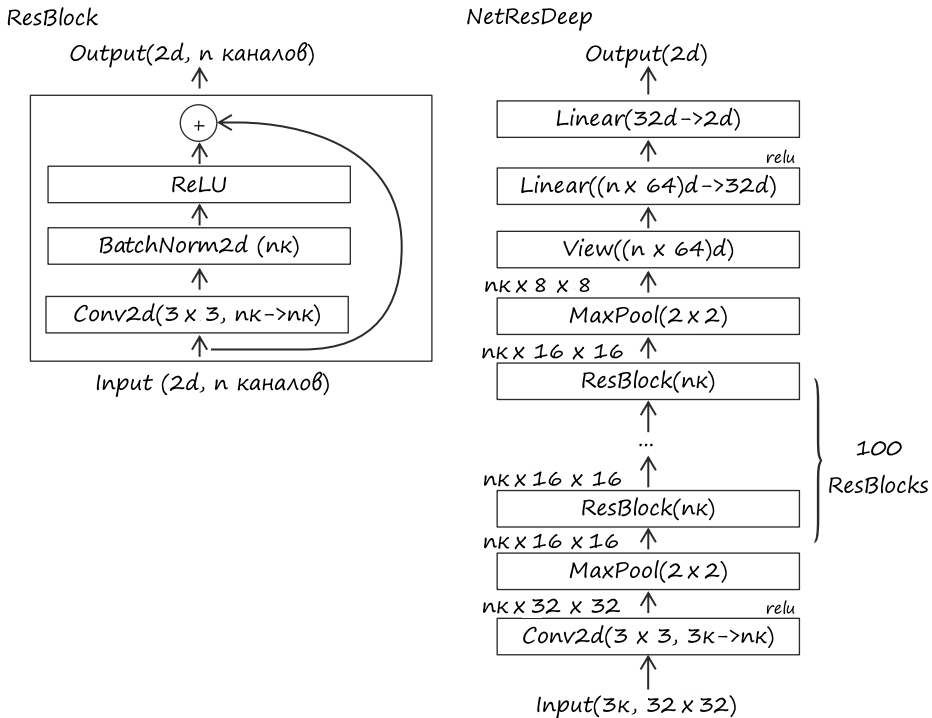


Рис. 8.12. Глубокая архитектура с остаточными связями. Слева изображен простой остаточный блок, который послужит стандартным блоком нашей сети, как показано справа

Поскольку мы хотим сгенерировать глубокую модель, то включаем в блок нормализацию по батчам, которая поможет предотвратить «исчезновение» градиентов во время обучения. Теперь мы хотели бы сгенерировать сеть из 100 блоков. Значит ли это, что нам нужно готовиться к утомительному копированию/вставке? Вовсе нет: у нас уже есть все необходимое, чтобы понять, как она будет выглядеть.

Прежде всего, в *init* мы создаем `nn.Sequential` со списком экземпляров `ResBlock`. `nn.Sequential` гарантирует, что выходной сигнал одного блока будет использован как входной сигнал следующего, а также что все параметры блока видимы `Net`. Далее мы просто вызываем в функции `forward` экземпляр `nn.Sequential` для обхода всех 100 блоков и получения результата:

```
# In[56]:
class NetResDeep(nn.Module):
    def __init__(self, n_chans1=32, n_blocks=10):
        super().__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
```

```

self.resblocks = nn.Sequential(
    *(n_blocks * [ResBlock(n_chans=n_chans1)]))
self.fc1 = nn.Linear(8 * 8 * n_chans1, 32)
self.fc2 = nn.Linear(32, 2)

def forward(self, x):
    out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
    out = self.resblocks(out)
    out = F.max_pool2d(out, 2)
    out = out.view(-1, 8 * 8 * self.n_chans1)
    out = torch.relu(self.fc1(out))
    out = self.fc2(out)
    return out

```

В реализации мы параметризуем количество слоев ради удобства переиспользования и проведения различных опытов. Конечно же, обратное распространение ошибки будет работать так, как и ожидалось. Неудивительно, что сходимость этой сети происходит намного медленнее, и сходимость эта более хрупкая. Именно поэтому мы тщательнее подобрали начальные значения и обучали нашу **NetRes** со скоростью обучения $3e-3$ вместо $1e-2$, как при обучении других наших сетей. Мы не обучали ни одну из них до стадии сходимости, но без этих уловок ничего бы не получилось.

Это не значит, что вам следует стремиться углублять сети для набора данных изображений 32×32 , но четко показывает, как сделать это для более сложных наборов данных наподобие ImageNet. А также предоставляет все необходимое для понимания уже существующих реализаций моделей наподобие ResNet, например, в **torchvision**.

Инициализация

Небольшой комментарий относительно вышеупомянутого задания начальных значений. Задание начальных значений — это один из важнейших приемов обучения нейронных сетей. К сожалению, по историческим причинам начальные значения по умолчанию для весовых коэффициентов в PyTorch неидеальны. Производятся активные попытки решить эту проблему; за их ходом можно следить на GitHub (<https://github.com/pytorch/pytorch/issues/18182>). Тем временем нам придется исправлять начальные значения весов самостоятельно. Мы обнаружили, что наша модель не сходится, и изучили распространенные варианты начальных значений (меньшая дисперсия весов, а также нулевое среднее значение и единичная дисперсия на выходе для нормализации по батчам), а затем уменьшили дисперсию выходного сигнала вдвое при нормализации по батчам, когда сеть не сходилась.

Инициализации весов можно посвятить целую главу, но нам это представляется излишним. В главе 11 мы снова столкнемся с вопросом задания начальных значений и воспользуемся вариантом, который вполне мог бы служить

в PyTorch значением по умолчанию без особых пояснений. Когда вы дорастете до того, что нюансы задания начальных значений весов начнут вас всерьез интересовать, — вероятно, не раньше чем дочитаете эту книгу, — можете вернуться к этому вопросу¹.

8.5.4. Сравнение архитектур этого раздела

Подытожим эффект от каждой из наших модификаций архитектуры по отдельности на рис. 8.13. Не стоит придавать слишком большое значение каким-либо

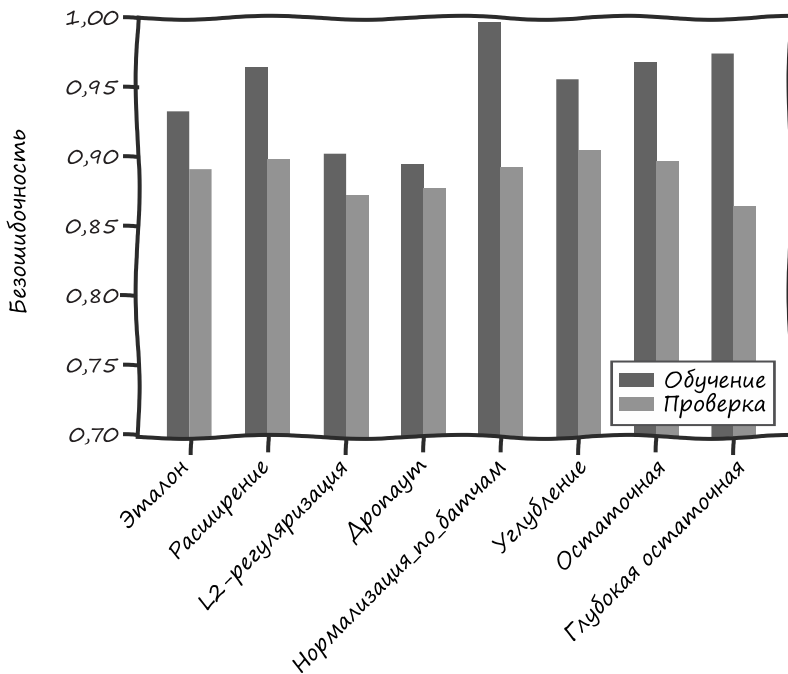


Рис. 8.13. Все модификации сетей дают схожие результаты

¹ Основополагающая статья на эту тему — авторства Х. Глорота (X. Glorot) и Й. Бенжио (Y. Bengio) — «Сложность обучения глубоких нейронных сетей прямого распространения» (Understanding the Difficulty of Training Deep Feedforward Neural Networks, 2010), в которой рассказывается о начальных значениях Хавьера PyTorch (<http://mng.bz/vxz7>). Уже упоминавшаяся статья ResNet развивает эту тему, представляя использовавшееся нами выше задание начальных значений Каймина. А позднее Х. Чжан (H. Zhang) и др. доработали технологию задания начальных значений до того, что смогли отказаться от нормализации по батчам в своих опытах с очень глубокими остаточными сетями (<https://arxiv.org/abs/1901.09321>).

конкретным числам — мы достаточно сильно упростили наши задачи и эксперименты, так что повторение конкретного эксперимента с различными случайными начальными значениями приведет, вероятно, к значительным различиям в точности проверки. Для этой демонстрации мы взяли все прочие элементы равными, от скорости обучения до количества эпох обучения; на практике мы пытались бы добиться наилучших результатов путем их изменения. Кроме того, вероятно, мы бы захотели присоединить некоторые из дополнительных архитектурных элементов.

Не помешают также количественные наблюдения: как мы видели в подразделе 5.5.3 при обсуждении проверки и переобучения, у регуляризаций в виде затухания весов и дропаута, у которых статистическая интерпретация в качестве методов регуляризации более строгая, чем нормализация по батчам, расхождение между интересующими нас двумя показателями безошибочности намного меньше. Нормализация по батчам, предназначенная скорее для улучшения сходимости, позволяет обучать сеть до степени безошибочности практически 100 %, так что мы будем интерпретировать два первых метода как методы регуляризации.

8.5.5. Описанное здесь уже устарело

Проклятие и благословение специалистов по глубокому обучению — чрезвычайно быстрая эволюция архитектуры нейронных сетей. Это не значит, что все обсуждавшееся в этой главе уже безнадежно устарело, но полноценная демонстрация новейших и лучших архитектур — задача уже другой книги (которая так же очень быстро перестанет быть новейшей и лучшей). Ключевой вывод: необходимо приложить все усилия к тому, чтобы научиться качественно переводить математические выкладки из научных статей в настоящий код PyTorch или по крайней мере понимать написанный другими с этой же целью код. Надеемся, что в последних нескольких главах вы обрели немало базовых навыков преобразования идей в реализованные на PyTorch модели.

8.6. ИТОГИ ГЛАВЫ

Мы проделали немалую работу и получили модель, с помощью которой наш вымышленный друг Джейн может фильтровать изображения для своего блога. Осталось взять входящее изображение, обрезать его и привести к размеру 32×32 , а затем узнать, что скажет о нем модель. Разумеется, мы решили только часть задачи, но и это немалое достижение.

А решили мы только часть задачи, потому что осталось несколько любопытных «белых пятен», с которыми нам пришлось бы столкнуться. Одно из них:

выделение птицы или самолета из большого изображения. Создание ограничивающих прямоугольников около объектов — пример того, на что модели, подобные нашим, не способны.

Еще одна трудность связана с тем, что происходит, когда перед камерой прогуливается соседский кот. Наша модель не сможет удержаться от высказывания мнения о степени птицеподобности кота! Она с радостью выдаст результат «птица» или «самолет», возможно даже с вероятностью 0,99. Подобная проблема излишней уверенности сети в классификации примеров данных, далеких от обучающего распределения, называется *чрезмерным обобщением* (*overgeneralization*) и является одной из основных проблем при вводе (вероятно, неплохой) модели в эксплуатацию в тех случаях, когда нельзя по-настоящему доверять входным данным (к сожалению, это большинство встречающихся на практике задач).

В этой главе мы создали приемлемые, работающие модели на PyTorch, способные обучаться на изображениях. И сделали это так, чтобы научиться лучше понимать сверточные сети. А также изучили, как можно расширить или углубить сеть, удерживая при этом под контролем такие эффекты, как переобучение. И хотя мы затронули лишь верхушку айсберга, но достаточно далеко продвинулись с предыдущей главы. Мы заложили прочный фундамент для решения сложных задач, с которыми мы столкнемся при работе над проектами глубокого обучения.

Теперь, познакомившись с соглашениями и основными возможностями PyTorch, мы готовы подступить к чему-то посерьезнее. Вместо того чтобы посвящать каждую главу или две небольшой задаче, мы собираемся потратить сразу много глав на решение большой реальной проблемы. Роль постоянного примера в главе 2 играет задача автоматического обнаружения рака легких; мы перейдем от знакомства с API PyTorch к реализации с его помощью целых проектов. Следующую главу мы начнем с общей постановки задачи, а затем опишем все нюансы используемых данных.

8.7. УПРАЖНЕНИЯ

1. Измените нашу модель, передав в конструктор `nn.Conv2d` аргумент `kernel_size=5`, чтобы использовать ядро размером 5×5 .
 - А. Как это изменение влияет на количество параметров модели?
 - Б. Усиливает или снижает переобучение это изменение?
 - В. Прочитайте <https://pytorch.org/docs/stable/nn.html#conv2d>.
 - Г. Можете ли вы описать, какой эффект окажет `kernel_size=(1,3)`?
 - Д. Как ведет себя модель с подобным ядром?

2. Сможете ли вы найти изображение, не содержащее ни птицы, ни самолета, на котором модель видит то или другое с вероятностью более 95 %?
 - А. Можете ли вы вручную отредактировать изображение так, чтобы сделать его более похожим на самолет?
 - Б. Можете ли вы так вручную отредактировать изображение самолета, чтобы модель ошиблась и сообщила, что это птица?
 - В. Упростятся ли эти задачи в случае сети с меньшими разрешающими возможностями?

8.8. РЕЗЮМЕ

- Свертки могут использоваться как линейные операции сети прямого распространения, обрабатывающей изображения. Благодаря сверткам получаются сети с меньшим числом параметров, использующие локальность и обладающие инвариантностью относительно сдвига.
- Размещение нескольких операций свертки друг за другом и использование максимального объединения между ними дает эффект применения сверток с учетом пространственных отношений в больших частях входного изображения по мере роста глубины.
- Все подклассы `nn.Module` могут рекурсивно собирать и возвращать параметры — свои и дочерние. Эту методику можно использовать для их подсчета, передачи оптимизатору или просмотра их значений.
- Функциональный API предоставляет модули, не зависящие от хранения внутреннего состояния. Он применяется для операций, где не хранятся параметры и, следовательно, не происходит обучения.
- Обученные параметры модели можно сохранить на диск, а затем загрузить обратно при помощи одной строки кода для каждой из этих операций.

Часть II

Обучение на изображениях на практике: раннее выявление рака легких

Часть II несколько отличается от части I, так как представляет собой практически еще одну книгу в книге. Мы возьмем одну задачу и подробно рассмотрим ее на протяжении нескольких глав. Начнем с основных строительных блоков, которые мы изучили в части I, и шаг за шагом создадим более полноценный проект, чем приведенный в примерах до этого. Первые варианты будут выдавать неполный и неточный результат, но мы рассмотрим, как найти и устранить возникающие проблемы. Мы также определим пути улучшения нашего решения, внедрим их и оценим степень их влияния. Для обучения моделей, которые мы разработаем в части II, вам потребуется графический процессор с оперативной памятью не менее 8 Гбайт, а также несколько сотен гигабайт свободного места на диске для хранения обучающих данных.

В главе 9 мы опишем задачу, среду и данные, которые будем использовать, а также структуру проекта, который предстоит реализовать. В главе 10 рассмотрим, как преобразовать наши данные в набор данных PyTorch. В главах 11 и 12 будет описана модель классификации, а именно метрики, с помощью которых будет оцениваться, насколько хорошо обучается набор данных. Кроме того, мы решим различные проблемы, затрудняющие обучение модели. В главе 13 начнем создание сквозного проекта и создадим модель сегментации, которая будет генерировать тепловую карту, а не одну классификацию. Эта тепловая карта будет использоваться для определения областей для классификации. Наконец, в главе 14 мы объединим созданные модели сегментации и классификации и поставим окончательный диагноз.



Применение PyTorch в борьбе с раком

В этой главе

- ✓ Разбиение большой задачи на более мелкие и простые составляющие.
- ✓ Ограничения непростой задачи глубокого обучения, структура и подход проекта.
- ✓ Загрузка обучающих данных.

В этой главе перед нами две основные цели. Для начала мы изложим общий план всей части II, чтобы иметь четкое представление о цельной картине, которую предстоит создать в последующих главах. В главе 10 мы начнем писать подпрограммы анализа и обработки данных; их результатом станут другие данные, которые в главе 11 будут использоваться при обучении нашей первой модели. Чтобы успешно выполнить все задачи в следующих главах, в этой мы рассмотрим общий контекст, в котором будет существовать наш проект. Мы рассмотрим форматы данных и их источники, а также исследуем ограничения, возникающие из предметной области. К этим задачам стоит привыкнуть, так как вам всегда предстоит выполнять их в рамках любого серьезного проекта глубокого обучения!

9.1. ПОСТАНОВКА ЗАДАЧИ

В этой части книги самое главное — овладеть инструментами, необходимыми в ситуациях, когда что-то не работает (а так бывает гораздо чаще, чем вы могли бы подумать, прочитав часть I). Невозможно предсказать каждый возможный

сбой или рассмотреть каждую технику отладки, но надеемся, что мы рассмотрим достаточно, чтобы неожиданно возникшее препятствие не стало для вас непреодолимым. Вдобавок мы хотим помочь вам избежать ситуаций, когда ваши собственные проекты работают неэффективно, но вы понятия не имеете, что делать. Вместо этого надеемся, что ваш список идей будет настолько длинным, что единственной сложностью будет выбрать нужную!

Чтобы представить эти идеи и методы, нам нужен контекст, причем довольно объемный и в меру сложный. Мы предлагаем решить задачу автоматического обнаружения злокачественных опухолей в легких, где в качестве входных данных имеется только компьютерная томография грудной клетки пациента. Мы сосредоточимся больше на технических проблемах, чем на факторах, связанных непосредственно с человеком, но это не должно расслаблять вас, ведь даже с чисто инженерной точки зрения успех в части II потребует более серьезного, структурированного подхода, чем тот, который мы использовали в части I.

ПРИМЕЧАНИЕ

КТ-сканы — это, по сути, трехмерные рентгеновские снимки, представленные в виде трехмерного массива одноканальных данных. Мы рассмотрим их более подробно чуть позже.

Как вы уже могли догадаться, название текущей главы бросается в глаза, гиперболизировано и несколько неточно излагает наши намерения. Уточним: в проекте в этой части книги в качестве входных данных мы будем использовать трехмерные компьютерные томограммы торса человека, а в качестве выходных — выдавать местонахождение предполагаемых злокачественных опухолей, если таковые будут найдены.

Раннее обнаружение рака легкого оказывает огромное влияние на шансы пациента выжить, но данную процедуру трудно осуществить вручную, особенно в масштабах всей популяции. В текущем варианте работа по анализу данных должна выполняться высококвалифицированными специалистами, требует кропотливого внимания к деталям, и в большинстве случаев рак не обнаруживается.

Для человека такая работа сродни тому, как если бы вас поставили перед сотней стогов сена и сказали: «Определите, в каких из них есть иголка». Выполняя подобную работу, легко пропустить опасный знак, особенно на ранних стадиях, когда эти знаки совершенно незаметны. Человеческий мозг просто не приспособлен для такой монотонной работы. Но тут на сцену выходит глубокое обучение.

Работа по автоматизации этого процесса позволит вам набраться опыта работы в сложных условиях, когда значительную часть работы придется выполнять с нуля, а простых ответов на возникающие вопросы будет все меньше. Но вместе мы добьемся результата! Проработав всю часть II, вы будете готовы приступить к любой реальной и еще не решенной проблеме, которую выберете сами.

Мы выбрали задачу обнаружения опухоли в легких по нескольким причинам. Главная заключается в том, что данная проблема еще не решена! Это важно, ведь мы хотим продемонстрировать, что вы можете использовать PyTorch для эффективного решения задач в самых передовых проектах. Мы надеемся, что это повысит вашу уверенность в PyTorch как в фреймворке, а также в себе как в разработчике. Хотя данная проблема пока не решена, в последнее время многие команды обратили на нее внимание и уже получили многообещающие результаты. Это означает, что решить ее, вероятно, мы можем уже в ближайшее время. Не будем в книге обсуждать проблемы, решение которых не появится в ближайшие десятилетия. Повышенное внимание к поставленной задаче также привело к появлению множества отличных статей и проектов с открытым исходным кодом, из которых можно почерпнуть вдохновение и хорошие идеи. Это станет для вас огромным подспорьем, если после изучения части II вы захотите улучшить проект, которому мы положим начало. В главе 14 мы приведем несколько ссылок на дополнительные материалы.

Эта часть книги, как вы уже поняли, посвящена задаче обнаружения опухолей в легких, но навыки, которые вы приобретете в процессе, носят общий характер. Умение исследовать предметную область, предварительно обрабатывать и создавать данные для обучения будет важно в работе над любым проектом. Задачу предварительной обработки мы будем решать в конкретном контексте опухолей легких, но общая мысль такова, что *обработкой предстоит заниматься в любом проекте*. Аналогично настройка цикла обучения, получение информативных показателей производительности и сборка моделей проекта в окончательное приложение — все это общие навыки, и именно их мы получим в главах с 9-й по 14-ю.

ПРИМЕЧАНИЕ

В конце части II мы получим готовый рабочий проект, но его результаты недостаточно точны для клинического использования. Эта задача должна мотивировать вас к изучению PyTorch, но мы не будем излагать все до единого инструменты и подходы для решения проблемы.

9.2. ПОДГОТОВКА К МАСШТАБНОМУ ПРОЕКТУ

В этом проекте вам понадобятся базовые навыки, полученные в части I. Особенно пригодится материал о построении модели, который приводился в главе 8. Большая часть нашей модели по-прежнему будет состоять из нескольких сверточных слоев с последующим слоем дискретизации и понижения разрешения. Но в качестве входных данных для нашей модели мы будем использовать 3D-данные. Концептуально они похожи на данные 2D-изображения, с которыми мы работали в последних нескольких главах части I, но ряд инструментов экосистемы PyTorch, специфичных для 2D, применить не получится.

Основная разница между работой, которую мы проделали со сверточными моделями в главе 8, и тем, что предстоит сделать в части II, заключается в объеме работы за пределами самой модели. В главе 8 мы использовали готовый набор данных и лишь немного подправили его, после чего ввели данные в модель для классификации. Тогда почти все наше время и внимание ушло на построение самой модели, а сейчас до главы 11 мы не собираемся начинать даже проектирование первой из двух архитектур. Дело в том, что в качестве входа у нас нестандартные данные, но нет готовых библиотек, которые сделали бы из входных обучающие данные, сразу готовые ко включению в модель. Нам придется самим изучить данные и реализовать обработку.

Но даже после этого будет недостаточно преобразовать КТ в тензор, передать его в нейронную сеть и дожидаться ответа с другой стороны. Как часто бывает в реальных задачах наподобие этой, работающий подход усложняется из-за необходимости учитывать дополнительные факторы, такие как ограниченная доступность данных, ограниченность вычислительных ресурсов и ограничение нашей способности разрабатывать эффективные модели. Это предстоит иметь в виду, когда мы будем описывать архитектуру нашего проекта на высоком уровне.

Кстати, об ограниченных вычислительных ресурсах: для работы с частью II нам будет нужен графический процессор, имеющий хотя бы 8 Гбайт ОЗУ, чтобы скорость обучения была приемлемой. Мы создадим такие модели, что их обучение с помощью ЦП может занять несколько недель!¹ Если у вас нет графического процессора, то можете прибегнуть к предварительно обученным моделям; их мы предоставим в главе 14. Там будет сценарий анализа конкреций, который, скорее всего, выполнится за ночь. Мы не хотим без необходимости опираться на проприетарные сервисы, но в данном случае придется. Сервис Collaboratory (<https://colab.research.google.com>) предоставляет бесплатные экземпляры графического процессора, которыми можно воспользоваться. А PyTorch там установлен из коробки! Для хранения необработанных обучающих данных, кэшированных данных и обученных моделей вам потребуется не менее 220 Гбайт свободного места на диске.

ПРИМЕЧАНИЕ

Во многих примерах кода, представленных в части II, опущены детали, усложняющие код. Мы не загромождаем примеры логированием, обработкой ошибок и граничных случаев и приводим лишь код, выражающий основную идею, о которой идет речь. Полные примеры рабочего кода можно найти на сайте книги (www.manning.com/books/deep-learning-with-pytorch) и в GitHub (<https://github.com/deep-learning-with-pytorch/dlwpt-code>).

Итак, мы поняли, что перед нами стоит трудная и многогранная задача. И что теперь со всем этим делать? Вместо того чтобы рассматривать всю КТ на наличие

¹ Это лишь предположение, на самом деле мы не пробовали.

признаков опухолей или их потенциальной злокачественности, мы решим ряд более простых задач, которые в совокупности позволят получить интересующий нас результат. Мы создадим нечто вроде конвейера: на каждом его этапе будут приниматься сырые данные и/или выходные данные с предыдущих этапов, затем будет выполняться некая обработка, а результат будет передаваться следующему этапу. Не каждую задачу нужно решать по такому шаблону, но любую задачу стоит разбить на отдельные части, которые можно реализовать по отдельности. Даже если для данного конкретного проекта подобный подход окажется не-правильным, во время работы над отдельными фрагментами вы можете узнать достаточно, чтобы преобразовать используемый подход в нечто более удачное.

Прежде чем мы подробно начнем говорить о разбиении задачи на части, нам нужно узнать некоторые подробности о предметной медицинской области. В листингах кодов будет написано, *что* мы делаем, но лишь теория радиационной онкологии объяснит, *почему* мы делаем именно так. Изучение проблемы с теоретической точки зрения имеет решающее значение, независимо от того, о какой области идет речь. Глубокое обучение — весьма действенный инструмент, но это не волшебная палочка. Попытки слепо применять данный инструмент к нетривиальным задачам, скорее всего, приведут к провалу. Вместо этого мы должны сочетать понимание проблемной области с интуитивными догадками о поведении нейронной сети. А далее последовательные эксперименты и уточнения должны дать нам достаточно информации, чтобы мы могли приблизиться к работоспособному решению.

9.3. ЧТО ТАКОЕ КОМПЬЮТЕРНАЯ ТОМОГРАФИЯ

Прежде чем мы углубимся в проект, нужно разобраться, что такое компьютерная томография. Именно ее данные будут основным форматом данных для нашего проекта, и понимание сильных и слабых сторон форматов данных и их фундаментальной природы будет иметь решающее значение для их правильного использования. Ключевой момент, который мы уже отмечали, заключается в следующем: компьютерная томография — это, по сути, трехмерные рентгеновские снимки, представленные в виде трехмерного массива одноканальных данных изображений. Из главы 4 мы помним, что такой формат похож на сложенный набор изображений PNG в оттенках серого.

ВОКСЕЛЬ

Воксель — трехмерный эквивалент привычного двумерного пикселя. Он занимает некий объем пространства (поэтому его называют «объемный пиксель»), а не плоскую область и обычно размещается в трехмерной сетке. Каждое из его измерений — это расстояние. Часто воксели имеют кубическую форму, но в текущей главе воксели будут представлять собой прямоугольные призмы.

Помимо медицинских данных, аналогичные воксельные данные используются в симуляциях жидкости, реконструкциях 3D-сцен из 2D-изображений, данных LIDAR у беспилотных автомобилей и во многих других проблемных областях. У каждой предметной области свои индивидуальные особенности и тонкости, и, хотя API, которые мы собираемся рассмотреть, написаны в достаточно общем виде, для эффективной работы с ними нужно понимать природу данных.

Каждый воксель КТ имеет числовое значение, которое примерно соответствует средней массовой плотности вещества, содержащегося в этой точке. На большинстве визуализаций подобных данных вещества высокой плотности, такие как кости и металлические имплантаты, отображаются белыми, воздух и легочная ткань с малой плотностью — черной, а жир и ткани — различными оттенками серого. В конечном итоге результат получается похожим на рентгеновский снимок, но с ключевыми различиями.

Основное различие между компьютерной томографией и рентгеновскими снимками заключается в том, что рентгеновский снимок представляет собой проекцию трехмерной интенсивности (в данном случае плотности ткани и костей) на двумерную плоскость, а компьютерная томография сохраняет данные в третьем измерении. Это позволяет нам отображать данные различными способами, например в виде твердого тела в градациях серого, как на рис. 9.1.

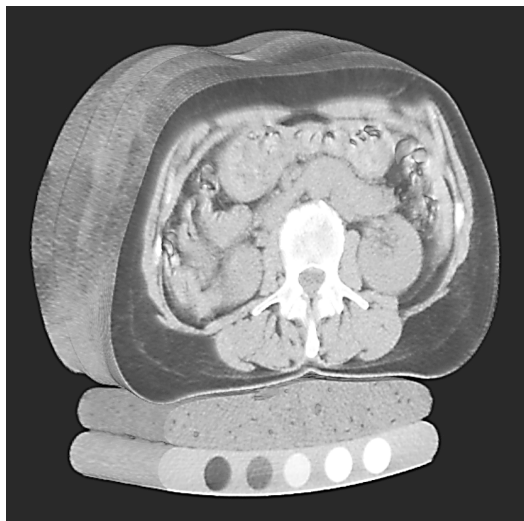


Рис. 9.1. КТ туловища человека, где видны (сверху вниз) кожа, органы, позвоночник и опорная койка пациента. Источник: <http://mng.bz/04r6>; Mindways CT Software / CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/deed.ru>)

ПРИМЕЧАНИЕ

Компьютерная томография фактически измеряет радиоплотность, которая является функцией как массовой плотности, так и атомного номера исследуемого вещества. Для наших целей это различие не имеет значения, поскольку модель будет потреблять данные КТ и учиться на них независимо от того, в каких единицах измерялись данные.

3D-представление также позволяет нам «заглянуть внутрь» объекта и скрывать типы тканей, которые нас не интересуют. Например, мы можем визуализировать данные в 3D и ограничить видимость только костной и легочной тканью, как показано на рис. 9.2.

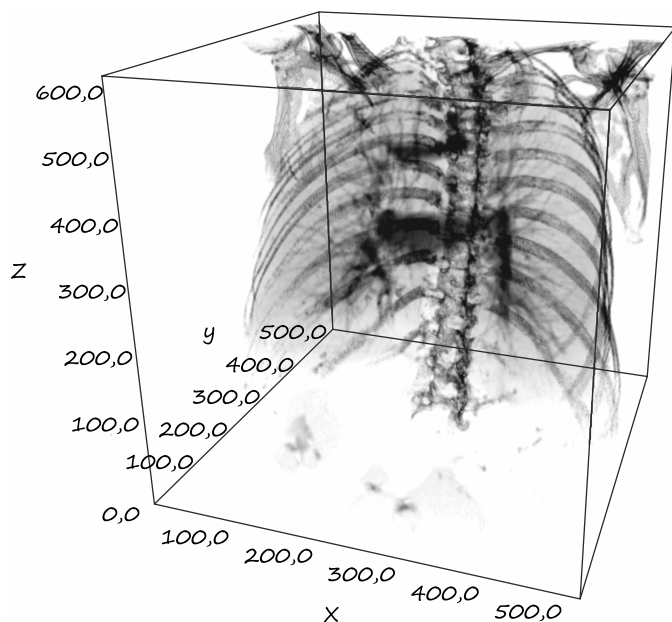


Рис. 9.2. КТ, на которой видны ребра, позвоночник и структура легких

Получить компьютерную томографию гораздо труднее, чем рентгеновские снимки, поскольку для этого требуется аппарат, показанный на рис. 9.3, а он обычно стоит более миллиона долларов и для работы с ним требуется обученный персонал. В большинстве больниц и некоторых хорошо оборудованных клиниках есть компьютерные томографы, но они далеко не так распространены, как рентгеновские аппараты. Это ограничение, в сочетании с врачебной тайной, может несколько затруднить получение снимков КТ, если только ранее кто-то уже не проделал работу по их сбору и организации.

На рис. 9.3 тоже приведен пример ограничивающей области при снятии КТ. Кровать, на которой лежит пациент, перемещается вперед-назад, позволяя сканеру получить несколько срезов пациента и, следовательно, заполнить ограничивающую рамку. Более темное центральное кольцо сканера — это место, где находится оборудование для обработки изображений.

Последнее различие между компьютерной томографией и рентгеном заключается в том, что данные КТ можно получить только в цифровом формате — https://en.wikipedia.org/wiki/CT_scan#Process.



Рис. 9.3. Пациент внутри КТ-сканера с подрисованной ограничивающей рамкой. Лишь на стоковых фотографиях пациенты лежат на томографии в уличной одежде

Необработанные данные после сканирования мало что говорят человеческому глазу и должны быть правильно преобразованы компьютером в понятную нам форму. Настройки КТ-сканера при выполнении сканирования могут значительно повлиять на итоговые данные.

Вся эта информация может показаться не особенно полезной, но это не так: из рис. 9.3 видно, что при сканировании компьютерным томографом расстояние по оси от головы до ног отличается от двух других осей. Пациент движется вдоль этой оси! Этот факт объясняет (или по крайней мере намекает на объяснение), почему наши воксели могут быть не кубическими, а также повлияет на наш подход к массивированию данных в главе 12. Это хороший пример того, почему понимание предметной области важно для выбора эффективного пути решения задачи. Приступая к работе над собственными проектами, вы должны проводить такое же исследование своих данных.

9.4. ПРОЕКТ: СКВОЗНОЙ ДЕТЕКТОР РАКА ЛЕГКИХ

Разобравшись с основами компьютерной томографии, теперь обсудим структуру нашего проекта. Большая часть места на диске уйдет на хранение 3D-массивов КТ-сканов, содержащих информацию о плотности тканей, а в моделях мы будем использовать различные части этих 3D-массивов. Разобьем задачу на пять основных этапов, последовательно двигаясь от исследования КТ всей грудной клетки до постановки диагноза рака легких у пациента.

Последняя задача особенно сложна, поскольку злокачественность новообразования не всегда очевидна из КТ, но посмотрим, чего удастся достичь. Наконец, совокупность классификаций узлов можно превратить в диагноз для пациента.

Распишем все этапы более подробно.

1. Преобразование необработанных данных компьютерной томографии в форму, с которой может работать PyTorch. Аналогичное преобразование всегда будет первым этапом в любом проекте, с которым вы столкнетесь. Этот процесс менее сложен, когда мы работаем с данными 2D-изображения, и еще больше упрощается при взаимодействии с данными, не являющимися изображениями.
2. Определение вокселей потенциальных опухолей в легких с помощью PyTorch и метода, известного как *сегментация*. Этот метод похож на создание тепловой карты областей, которые должны быть загружены в наш классификатор на этапе 3. Это позволит нам проанализировать потенциальные опухоли внутри легких и игнорировать не интересующие нас части тела (у человека точно не может быть рака легких в желудке).

В целом способность сосредоточиться на одной небольшой задаче во время обучения весьма полезна. С опытом возникают ситуации, когда более сложные структуры моделей могут давать превосходные результаты (например, игра GAN, которую мы видели в главе 2), но их проектирование с нуля требует в первую очередь искусного владения основными строительными блоками. Сначала нужно научиться ходить и лишь потом — бегать.

3. Группировка интересующих нас вокселей в области, а именно узелки-кандидаты (подробнее об узелках см. рис. 9.5 ниже). Мы должны будем приблизительно определить центр каждой целевой точки на тепловой карте.

Каждый узелок можно определить по его индексу, то есть строке и столбцу его центральной точки. Это делается для того, чтобы упростить и ограничить задачу перед передачей данных финальному классификатору. Группировка вокселей не будет задействовать PyTorch напрямую, так что мы вынесли это в отдельный этап. Часто при работе с многоступенчатыми решениями и переключении между большими частями проекта с глубоким обучением используются промежуточные этапы, не связанные с глубоким обучением.

4. Подтверждение или опровержение того, что узелок-кандидат и в самом деле является таковым, с помощью 3D-свертки.

Принцип реализации будет похож на двумерную свертку, которую мы рассмотрели в главе 8. Признаки, по которым мы определяем природу опухоли по структуре кандидата, являются локальными для рассматриваемой опухоли. Как следствие, этот подход должен обеспечить хороший баланс между ограничением размера входных данных и исключением релевантной информации. Применение ограничивающих объем подходов позволяет сократить

выполнение каждой отдельной задачи и количество вещей, которые придется изучать при устранении неполадок.

5. Постановка диагноза с помощью комбинированной классификации узелков.

Подобно классификатору узелков на предыдущем этапе, мы попытаемся определить доброкачественность или злокачественность узла, имея на руках только данные визуализации. Мы возьмем простой максимум прогнозов злокачественности для каждой опухоли, поскольку для диагностирования достаточно одной злокачественной опухоли. В других проектах, возможно, будет лучше использовать другие способы агрегирования прогнозов для каждого экземпляра в файл. Здесь мы спрашиваем: «Есть ли что-нибудь подозрительное?» — и для агрегации используем максимум. Если бы мы искали количественную информацию, такую как «отношение ткани типа А к ткани типа В», то могли бы вместо этого взять подходящее среднее значение.

На рис. 9.4 показан только окончательный путь данных через систему, когда мы уже создали и обучили все необходимые модели. А сама работа, которая требуется для обучения моделей, будет подробно описана по мере приближения к реализации каждого этапа.

Данные, которые мы будем использовать для обучения, представляют собой описанные человеком выходные данные для этапов 3 и 4. Это позволяет нам рассматривать этапы 2 и 3 (идентификация вокселей и их группировка на узелки-кандидаты) почти как проект, отдельный от этапа 4 (классификация кандидатов).

НА ПЛЕЧАХ ГИГАНТОВ

Выбирая описанный подход из пяти этапов, мы становимся похожи на карликов, стоящих на плечах гигантов. Об этих гигантах и их работе мы еще поговорим в главе 14. Сейчас нам не особенно нужно заранее знать, хорошо ли такая структура проекта подойдет для данной задачи. Вместо этого мы полагаемся на опыт других людей, которые уже реализовали подобные вещи и пришли к успеху. В других задачах вам придется экспериментировать, чтобы найти работающие подходы, но всегда важно стараться учиться на более ранних работах в этой области и у тех, кто уже занимался аналогичными задачами и обнаружил нечто полезное. Посмотрите, чего уже достигли другие, и используйте это как эталон. В то же время избегайте слепого копирования и запуска кода, поскольку вам нужно полностью понимать код, который вы запускаете, чтобы применять результаты для собственного прогресса.

Эксперты аннотировали данные, указав местоположение узелков конкреций, поэтому мы можем работать над этапами 2 и 3 или над этапом 4 в любом порядке, который нам нравится.

Сначала мы поработаем над этапом 1 (загрузка данных), затем перейдем к этапу 4, а далее вернемся к этапам 2 и 3, поскольку этап 4 (классификация) требует подхода, аналогичного использованному нами в главе 8, — множественных сверточных вычислений и объединения слоев для агрегирования пространственной информации перед ее подачей в линейный классификатор. Как только мы разберемся с нашей моделью классификации, можем приступить к этапу 2 (сегментация). Поскольку это более сложная тема, мы хотим заняться ею, не сосредотачиваясь одновременно на сегментации и основах компьютерной томографии и диагностике злокачественных опухолей. Вместо этого мы будем исследовать область обнаружения рака, работая над более знакомой задачей классификации.

Тот факт, что мы начинаем с середины задачи, вероятно, кажется странным. Начать с этапа 1 и продвигаться вперед было бы более интуитивно понятно. Однако возможность разделить проблему и работать над этапами независимо друг от друга полезна, поскольку позволяет вырабатывать более модульные решения. Так легче разделить рабочую нагрузку между членами небольшой команды. Кроме того, фактические клинические пользователи, вероятно, предпочтут систему, которая будет отмечать подозрительные узелки для последующей проверки, а не давать единый бинарный диагноз. Адаптировать наше модульное решение к различным вариантам использования, вероятно, будет проще, чем если бы мы сделали монолитную систему.

По мере работы над реализацией каждого этапа мы будем подробно рассказывать об опухолях легких, а также приводить подробности о работе компьютерной томографии. Подобный разговор может показаться не соответствующим теме книги, посвященной PyTorch, но мы намеренно уделяем этому время, чтобы вы начали размышлять в терминах предметной области. Это очень важный навык, поскольку пространство всех возможных решений и подходов слишком велико, чтобы эффективно писать код, выполнять обучение и проводить оценку.

Если бы мы работали над другим проектом (скажем, над тем, за который вы взялись после прочтения этой книги), то нам все равно пришлось бы провести исследование, чтобы понять данные и проблемную область. Возможно, вы интересуетесь спутниковой картографией и в следующем проекте будете использовать снимки планеты, сделанные с орбиты. Вам нужно задать вопросы о собираемых длинах волн — вы получаете только обычный RGB-спектр или что-то более экзотическое? А как насчет инфракрасного или ультрафиолетового спектра? Кроме того, на изображения может влиять время суток или положение объекта на изображении не прямо под спутником, что может привести к перекосу изображения. Потребуется ли коррекция?

Даже если тип данных вашего гипотетического *третьего* проекта останется прежним, вполне вероятно, что предметная область, в которой вы будете работать, кардинально изменит ситуацию. Обработка данных с камеры для беспилотных автомобилей по-прежнему подразумевает работу с 2D-изображениями, но сложности

тут совершенно другие. Например, маловероятно, что картографический спутник будет беспокоиться о попадании солнца в камеру или грязи на объектив!

Мы должны уметь пользоваться интуицией и направлять усилия на исследование потенциальных оптимизаций и улучшений. Это справедливо для проектов глубокого обучения в целом, и мы будем практиковаться в использовании интуиции, изучая часть II. Итак, поехали. Вернитесь на шагок назад и обратитесь к внутреннему взору. Что ваша интуиция говорит об этом подходе? Не кажется ли он слишком сложным?

9.4.1. Почему нельзя просто передавать данные в нейронную сеть, пока она не заработает

Прочитав последний раздел, вы могли вполне резонно подумать: «Это совсем не похоже на главу 8!» Вам может быть интересно, откуда взялись отдельные архитектуры моделей или почему общий поток данных такой сложный. Наш подход отличается от подхода, описанного в главе 8, по одной причине: для автоматизации это сложная задача и люди разобрались в ней еще не до конца. Но эта трудность сойдет на нет, когда мы, общество, окончательно решим данную проблему; скорее всего, появится готовый пакет библиотек, который будет работать из коробки, но пока мы этого не сделали.

Почему же это так сложно?

Что ж, для начала, большинство КТ не несут ничего интересного в плане ответа на вопрос: «Есть ли у данного пациента злокачественная опухоль?» Это и понятно, поскольку большая часть тела пациента будет состоять из здоровых клеток. При наличии злокачественной опухоли до 999 999 % вокселей на КТ все равно не будут раком. Это соотношение эквивалентно двухпиксельному пятну неправильно окрашенного цвета на HD-видео или одной грамматической ошибке в целом сборнике романов.

Можете ли вы найти белую точку на трех проекциях рис. 9.5, которая была помечена как узелок?¹

Если вам нужна подсказка, то можно использовать значения индекса, строки и столбца, чтобы помочь найти соответствующую каплю плотной ткани. Как вы думаете, сможете ли вы определить свойства опухолей, имея лишь изображения (а это означает *только* изображения — без информации об индексах, строках и столбцах!) вроде этих? Что, если бы вам дали весь 3D-скан, а не только три среза, пересекающие интересующую часть скана?

¹ series_uid для этого образца — 1.3.6.1.4.1.14519.5.2.1.6279.6001.126264578931778258890371755354, что может быть полезно, если позже вы захотите рассмотреть его подробнее.

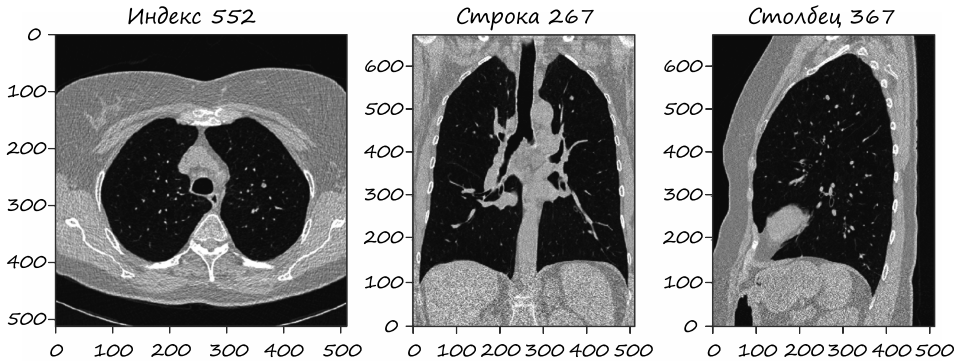


Рис. 9.5. Компьютерная томография, на которой видны примерно 1000 структур, для неопытного глаза выглядящие как опухоли. При осмотре специалистом-человеком был найден один узелок, а остальные являются нормальными анатомическими структурами, такими как кровеносные сосуды, поражения и другие непроблемные образования

ПРИМЕЧАНИЕ

Не волнуйтесь, если не можете определить местонахождение опухоли! Мы пытаемся проиллюстрировать, насколько сложными для анализа могут быть эти данные, и то, что их трудно идентифицировать визуально, — вот смысл приведенного примера.

Возможно, вы где-то видели, что сквозные подходы к обнаружению и классификации объектов очень успешно работают в задачах компьютерного зрения. В TorchVision есть сквозные модели, такие как Fast R-CNN/Mask R-CNN, но они обычно обучаются на сотнях тысяч изображений, и эти наборы данных не ограничены неким количеством примеров из редких классов. Преимущество архитектуры проекта, которую мы будем использовать, заключается в том, что она хорошо работает с более скромным объемом данных. Таким образом, теоретически мы можем передать в нейронную сеть сколь угодно большое количество данных. Однако пока она не изучит специфику пресловутой потерянной иглы, а также то, как игнорировать сено, будет практически невозможно собрать достаточно данных и правильно обучить сеть. Это будет не *лучший* подход, поскольку результаты получатся плохие и у большинства читателей не будет доступа к вычислительным ресурсам, чтобы вообще это осуществить.

Чтобы найти наилучшее решение, мы могли бы исследовать проверенные модели для сквозного анализа данных¹. Эти сложные конструкции могут давать высококачественные результаты, но они не самые *лучшие*, поскольку

¹ Например, Retina U-Net (<https://arxiv.org/pdf/1811.08661.pdf>) и сеть FishNet (<http://mng.bz/K240>).

для понимания проектных решений, стоящих за ними, необходимо сначала овладеть фундаментальными концепциями. Как следствие, продвинутые модели становятся плохими кандидатами для использования при изучении этих самых концепций!

Это не означает, что наш многоступенчатый дизайн является *лучшим* подходом, но это потому, что определение «лучший» зависит от критериев, выбранных для оценки подходов. Существует *множество* «лучших» подходов, как и множество целей, которые можно преследовать, работая над проектом. Наш автономный, многоступенчатый подход также имеет ряд недостатков.

Вспомните игру GAN из главы 2. Там у нас было две сети, работавшие вместе для создания убедительных подделок работ старых мастеров. Художник создавал работу-кандидата, а ученый критиковал ее, давая художнику обратную связь о том, как улучшить подделку.

С технической точки зрения структура модели позволяла градиентам распространяться от окончательного классификатора (фальшивого или настоящего) к самым ранним частям проекта (исполнителю).

Наш подход к решению проблемы не будет использовать сквозное обратное распространение градиента для прямой оптимизации конечной цели. Вместо этого мы будем оптимизировать отдельные фрагменты задачи по отдельности, поскольку наша модель сегментации и модель классификации не будут обучаться в тандеме. Такой прием может ограничить максимальную эффективность нашего решения, но мы считаем, что он значительно улучшит процесс обучения.

Мы чувствуем, что возможность думать об одном этапе одновременно позволяет нам увеличивать масштаб и изучать меньше нового за раз. Каждая из наших двух моделей будет сосредоточена на выполнении ровно одной задачи. Подобно человеку-рентгенологу, просматривающему КТ срез за срезом, обучение тоже легче проводить, если область хорошо ограничена. Мы также хотим предоставить инструменты, которые позволяют манипулировать данными. Возможность увеличивать масштаб и фокусироваться на деталях в определенной области окажет огромное влияние на общую производительность при обучении модели по сравнению с необходимостью одновременного просмотра всего изображения. Наша модель сегментации вынуждена потреблять весь скан, но мы структурируем все так, что наша классификационная модель будет получать увеличенное изображение интересующих областей.

На этапе 3 (группировка) мы получим и передадим на этап 4 (классификация) данные, аналогичные изображениям, приведенным на рис. 9.6, на котором показаны последовательные поперечные срезы опухоли. Это изображение представляет собой крупный план (потенциально злокачественной или по крайней мере неопределенной) опухоли, и модель на этапе 4 должна будет

научиться идентифицировать такие опухоли, а модель этапа 5 — классифицировать их как доброкачественные или злокачественные. Это образование для нетренированного глаза (или необученной сверточной сети) может показаться совершенно непонятным, и выявление признаков злокачественного новообразования в этом образце — гораздо более сложная проблема, чем работа с полным КТ-сканом, о которой мы говорили ранее. В коде следующей главы мы опишем процедуры создания увеличенных изображений узелков, подобных рис. 9.6.

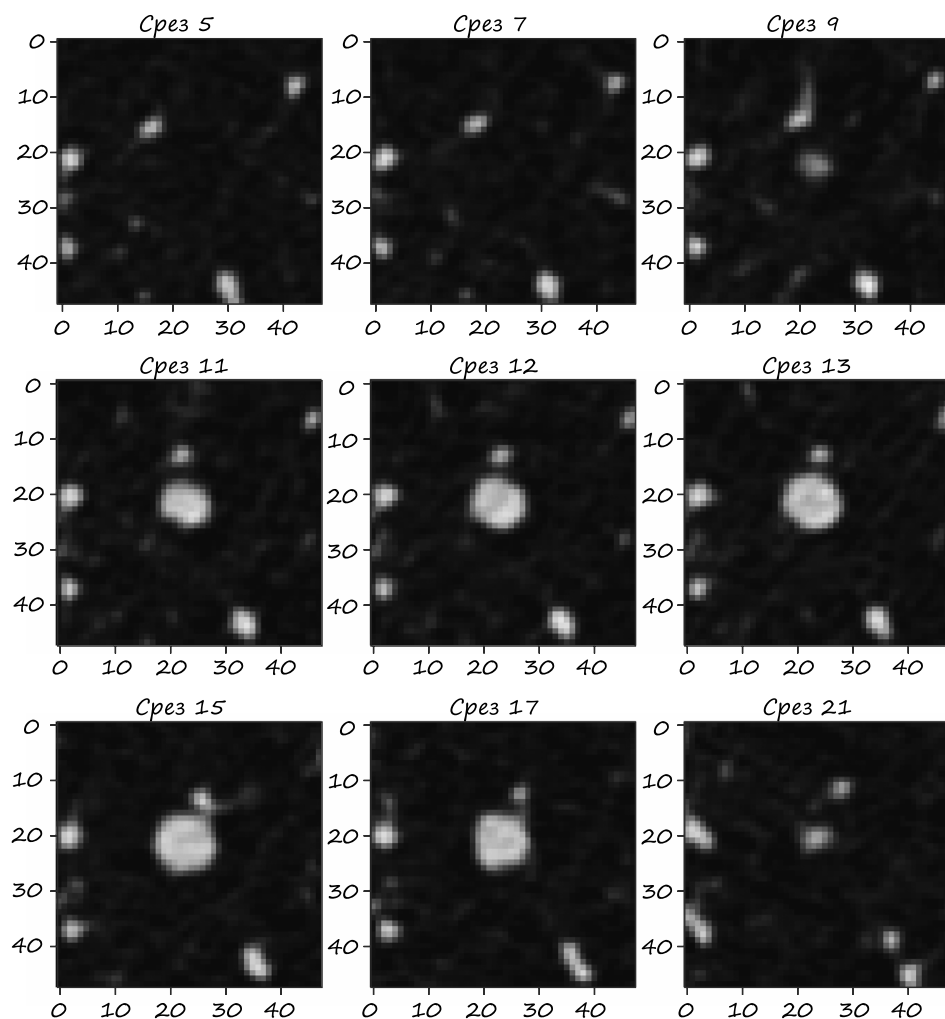


Рис. 9.6. Крупный план мультисреза опухоли, полученного при компьютерной томографии, приведенной на рис. 9.5

9.4.2. Что такое узелок

Как мы уже говорили, нам нужно уяснить некоторые особенности рака и радиационной онкологии, чтобы достаточно хорошо понять данные и использовать их эффективно. Последняя ключевая вещь, которую нам нужно понять, — что такое *узелок*. Говоря простым языком, узелок — это любой из множества шишек и образований, которые могут появляться в легких человека. Одни из них негативно сказываются на здоровье пациента, другие — нет. Точное определение¹ узелка ограничивает его размер тремя сантиметрами или меньше, при этом более крупная шишка уже называется *массой легкого*. Однако мы будем использовать понятие *узелка* взаимозаменяемо для всех схожих анатомических структур, поскольку граница в 3 см довольно условная, и станем обрабатывать тем же кодом и более крупные образования. Итак, узелок — небольшое образование в легком — может оказаться доброкачественной или злокачественной опухолью (также называемой *раком*). С радиологической точки зрения узелок похож на другие виды образований: это могут быть воспаления, инфекции, проблемы с кровотоком, видоизмененные кровеносные сосуды и другие опухоли.

Ключевым является следующее: рак, который мы пытаемся обнаружить, всегда будет узелком, подвешенным в неплотной ткани легкого или прикрепленным к его стенке. Это означает, что мы можем ограничить наш классификатор только узелками, а не исследовать всю ткань. Возможность ограничить объем входных данных поможет нашему классификатору лучше изучить поставленную задачу.

Это еще один пример того, что лежащие в основе методы глубокого обучения, которые мы будем использовать, универсальны, но их нельзя применять вслепую². Важно понимать область, в которой мы работаем, чтобы принимать правильные решения.

На рис. 9.8 показан типовой пример злокачественного узла. Наименьшие узелки, которые нас будут интересовать, имеют диаметр всего несколько миллиметров, хотя на рис. 9.8 показан более крупный образец. Как мы говорили ранее в этой главе, самые маленькие узлы примерно в миллион раз меньше. Больше половины узлов, выявляемых у пациентов, не злокачественные³.

¹ Olson E.J. Lung nodules: Can they be cancerous? Mayo Clinic. <http://mng.bz/yyge>.

² По крайней мере, если хотим получить хороший результат.

³ Согласно терминологии Национального института исследования рака: <http://mng.bz/jgBP>.

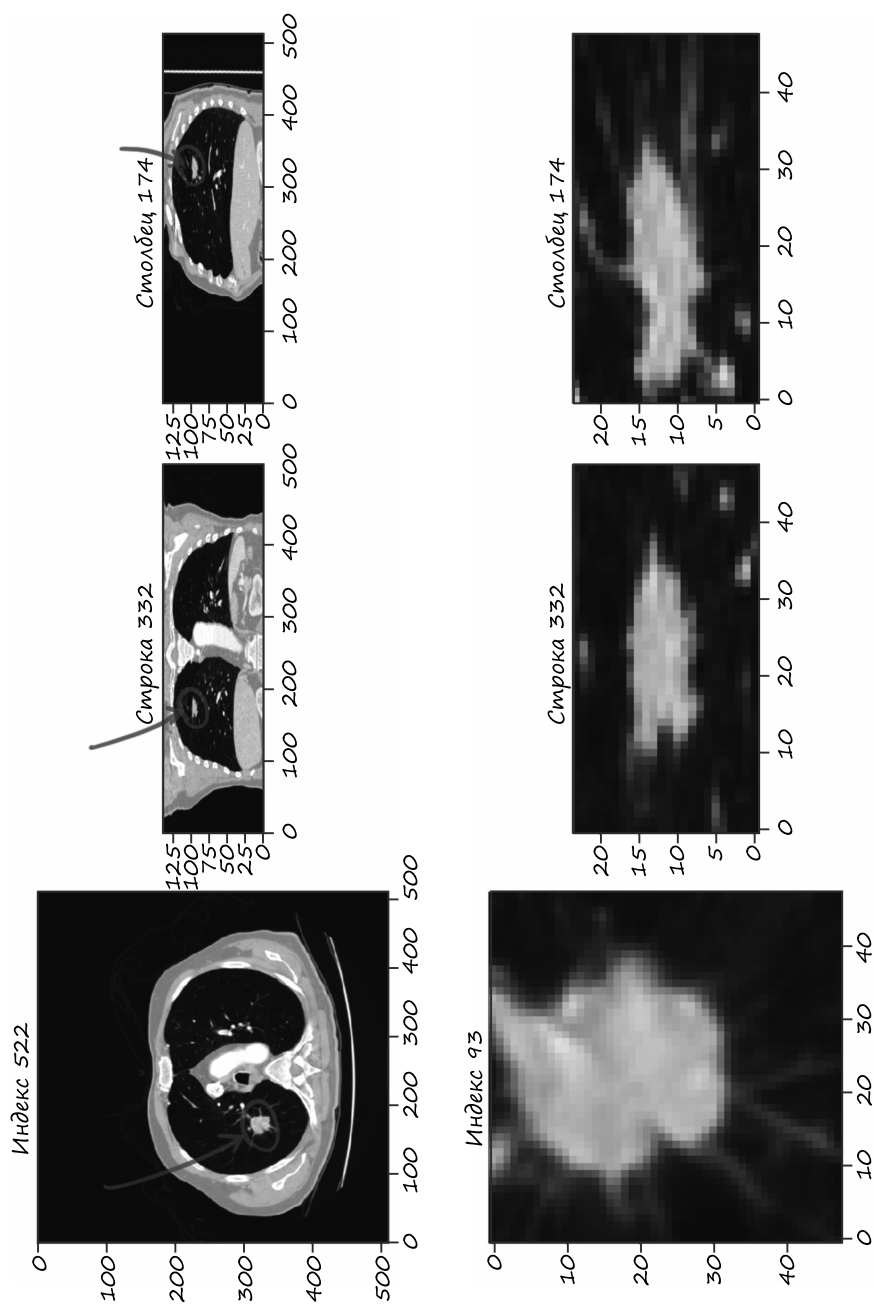


Рис. 9.8. Компьютерная томография злокачественного узелка, не похожего на другие

9.4.3. Наш источник данных: The LUNA Grand Challenge

Компьютерная томография, которую мы только что рассматривали, была получена в рамках конкурса LUNA (LUng Nodule Analysis). LUNA Grand Challenge — это сочетание работы с открытым набором данных, качественных меток КТ пациентов (часто с узелками в легких) и публичного ранжирования классификаторов по этим данным. Существует нечто вроде культуры публичного обмена наборами медицинских данных для исследований и анализа, и открытый доступ к таким данным позволяет исследователям выполнять новые работы с этими данными и пользоваться результатами старых работ, избегая необходимости заключать официальные соглашения об исследованиях между учреждениями (очевидно, что некоторые данные также хранятся в тайне). LUNA Grand Challenge ставит целью поощрить разработку новых методов обнаружения узелков и облегчить для команд борьбу за высокие позиции в таблице лидеров. Команда проекта может проверить эффективность своих методов обнаружения по стандартизированным критериям (на готовом наборе данных). Чтобы быть включенной в публичный рейтинг, команда должна предоставить научную работу с описанием архитектуры проекта, методов обучения и т. д. Это отличный ресурс для дальнейших идей и вдохновения для улучшения проекта.

ПРИМЕЧАНИЕ

Многие КТ-сканы «в дикой природе» значительно отличаются друг от друга с точки зрения особенностей различных моделей сканеров и программ обработки. Например, часть сканеров указывают на области КТ-скана, которые находятся за пределами поля зрения сканера, устанавливая отрицательную плотность вокселей в этих местах. У попадающих вам в руки сканов также могут быть различные настройки КТ-сканера, что тоже может повлиять на итоговое изображение как слегка, так и значительно. Хотя данные LUNA в целом чистые, при использовании других источников данных нужно обязательно проверять подобные моменты.

Мы будем использовать набор данных LUNA 2016. На сайте LUNA (<https://luna16.grand-challenge.org/Description>) описаны две задачи: первая — Nodule detection (NDET) — примерно соответствует нашему этапу 1 (сегментация); а вторая — False positive reduction (FPRED) — аналогична нашему этапу 3 (классификация). Когда на сайте говорят о «местоположении возможных узелков», речь идет о процессе, который мы рассмотрим в главе 13.

9.4.4. Загрузка данных LUNA

Прежде чем углубиться в детали проекта, расскажем, где взять данные, которые мы будем использовать. Это около 60 Гбайт сжатых данных, так что

в зависимости от скорости вашего интернета скачивание может занять некоторое время. После распаковки потребуется около 120 Гбайт места. И нам понадобится еще около 100 Гбайт места в кэше для хранения небольших фрагментов данных, чтобы обращаться к этим данным можно было быстрее, не читая всю КТ¹.

Перейдите по ссылке <https://luna16.grand-challenge.org/download> и либо зарегистрируйтесь с помощью электронной почты, либо используйте логин Google OAuth. После входа в систему вы увидите две ссылки для скачивания данных Zenodo, а также ссылку на Academic Torrents. В обоих случаях данные будут одинаковыми.

СОВЕТ

На момент написания этой книги на сайте luna16.grand-challenge.org нет ссылок на страницу скачивания данных. Если у вас возникли проблемы с поиском этой страницы, то проверьте, что в адресе написано именно `luna16.`, а не `luna.`, и при необходимости повторно введите URL.

Данные, которые мы будем использовать, состоят из десяти подмножеств, помеченных от `subset0` до `subset9`. Разархивируйте их все в отдельные подкаталоги, такие как `code/data-unversioned/part2/luna/subset0` и т. д. В Linux вам понадобится утилита декомпрессии `7z` (в Ubuntu ее можно скачать через пакет `p7zip-full`). Пользователи Windows могут скачать архиватор с веб-сайта 7-Zip (www.7-zip.org). Некоторые утилиты распаковки не умеют открывать архивы. Если вы получаете сообщение об ошибке, то проверьте, что у вас полная версия экстрактора.

Кроме того, вам потребуются файлы `candidates.csv` и `annotations.csv`. Для удобства мы разместили эти файлы на сайте книги и в репозитории GitHub, поэтому они уже должны лежать в файле `code/data/part2/luna/*.csv`. Их также можно скачать из того же места, что и подмножество данных.

ПРИМЕЧАНИЕ

Если у вас свободно менее 220 Гбайт на диске, то можно запустить примеры, используя только одно или два из десяти подмножеств данных. Уменьшение обучающего набора приведет к тому, что модель будет работать намного хуже, но это лучше, чем вообще ничего не запустить.

Как только вы скачаете файл кандидатов и по крайней мере одно подмножество, распакуете их и поместите в нужное место, можно будет приступить к выполнению примеров из этой главы. Если хотите забежать вперед, то можете

¹ Необходимое количество свободного места в кэше оговаривается в каждой главе, и как только закончите изучение главы, вы можете очистить кэш, чтобы освободить место.

использовать код `/p2ch09_explore_data.ipynb` из Jupiter Notebook. Если нет, то мы вернемся к коду и рассмотрим его более подробно позже. Надеемся, вам удастся скачать файлы до того, как вы начнете читать следующую главу!

9.5. ИТОГИ ГЛАВЫ

Мы сделали важный шаг в реализации нашего проекта! У вас может возникнуть ощущение, что мы не сделали ничего особенного, ведь ни одной строчки кода еще не написано. Но важно помнить, что в своих собственных проектах вам будет необходимо проводить исследование и подготовку; собственно, мы это и сделали.

В этой главе мы:

- лучше поняли контекст нашего проекта по теме обнаружения рака легких;
- наметили направление и структуру нашего проекта для части II.

Если вы все еще чувствуете, что мы не достигли никакого реального прогресса, то отриньте эти мысли, поскольку понимание пространства, в котором работает ваш проект, имеет решающее значение, а работа по проектированию, проведенная заранее, щедро окупится далее. Вскоре, как только мы начнем реализовывать наши процедуры загрузки данных в главе 10, выгода станет очевидна.

Так как эта глава носила информационный характер и в ней не было кода, мы пока обойдемся без упражнений.

9.6. РЕЗЮМЕ

- Наш подход к обнаружению раковых узелков будет состоять из пяти крупных этапов: загрузки данных, сегментации, группировки, классификации, анализа узелков и постановки диагноза.
- Разбиение крупного проекта на более мелкие полунезависимые подпроекты облегчает обучение. Для других проектов с другими целями более подходящими могут оказаться другие подходы.
- Компьютерная томография представляет собой трехмерный массив данных интенсивности, содержащий около 32 миллионов вокселей, что примерно в миллион раз больше, чем узелки, которые нам надо будет распознавать. Работа модели с отдельным кадром КТ в рамках задачи облегчит получение разумных результатов от обучения.

- Понимание природы данных облегчает написание процедур их обработки и позволяет не уничтожить важные части данных. Массив данных компьютерной томографии обычно не содержит кубических вокселей, а для отображения информации о местоположении в физических единицах измерения в индексы массива требуются преобразования. Интенсивность точек на компьютерной томографии примерно соответствует массовой плотности тканей, но в уникальных единицах измерения.
- Определение ключевых концепций проекта и правильная их композиция невероятно важны. В основном весь проект будет вращаться вокруг узелков, представляющих собой небольшие легочные образования, которые можно обнаружить на КТ. В легких бывают и другие структуры, имеющие похожий вид.
- Для обучения нашей модели мы будем использовать данные LUNA Grand Challenge. Они содержат снимки компьютерной томографии, а также аннотированные человеком выходные данные для классификации и группировки. Наличие качественных данных весьма важно для успеха проекта.

10

Объединение источников данных

В этой главе

- ✓ Загрузка и обработка файлов необработанных данных.
- ✓ Реализация на Python классов для представления данных.
- ✓ Преобразование данных в формат, используемый PyTorch.
- ✓ Визуализация процессов обучения и проверки данных.

С общими целями для части II мы уже разобрались, а заодно наметили, как данные будут проходить через нашу систему. Теперь углубимся в детали того, что мы собираемся делать в этой главе. Пришло время реализовать базовые процедуры загрузки и обработки необработанных данных. По сути, в любом серьезном проекте, над которым вам предстоит работать, нужно будет делать нечто аналогичное¹. На рис. 10.1 показана общая карта нашего проекта из главы 9. А в целом в этой главе мы сосредоточимся на этапе 1 — на загрузке данных.

Наша цель — научиться создавать обучающую выборку, используя входные необработанные данные КТ и их аннотации. Задача может показаться простой, но для того, чтобы загружать, обрабатывать и извлекать требуемые данные, нужно еще кое-что сделать. На рис. 10.2 показано, что нужно сделать, чтобы превратить необработанные данные в обучающий набор. К счастью, в предыдущей главе мы уже обрели некоторое *понимание* наших данных, но работы здесь еще немало.

¹ Если вы тот редкий исследователь, который заранее подготовил для себя все свои данные, то вам повезло! Остальные будут заняты написанием кода для загрузки и синтаксического анализа.

Это невероятно важный момент, ведь мы начинаем превращать сырые данные если не в золото, то по крайней мере в то, что наша нейронная сеть точно превратит в золото. Мы впервые обсудили механику этого преобразования в главе 4.

10.1. ФАЙЛЫ НЕОБРАБОТАННЫХ ДАННЫХ КТ

Данные КТ у нас представлены в двух видах файлов: в файлах `.mhd`, содержащих метаданные заголовков, и файлах `.raw`, содержащих необработанные байты, в виде трехмерных массивов. Имя каждого файла начинается с уникального идентификатора, называемого *UID* (название происходит от номенклатуры цифровых изображений и коммуникаций в медицине (Digital Imaging and Communications in Medicine, DICOM) для компьютерной томографии. Например, UID 1.2.3 соответствуют два файла: `1.2.3.mhd` и `1.2.3.raw`.

Наш класс `Ct` будет использовать два этих файла и создавать трехмерный массив, а также матрицу преобразования из системы координат пациента (которую мы обсудим более подробно в разделе 10.6) в координаты индекса, строки и столбца, необходимые для массива (эти координаты на рисунках называются (I, R, C), а у переменных в коде добавляется суффикс `_irc`). Прямо сейчас не нужно вдаваться в детали всего этого. Просто помните, что мы выполняем некое преобразование систем координат, чтобы затем применить эти координаты к нашим данным КТ. Подробности рассмотрим по мере необходимости.

Мы также загрузим данные аннотаций, предоставленные LUNA, в которых приведен список координат узелков с информацией об их злокачественности, а также UID соответствующего скана КТ. Комбинируя координату узла с информацией о преобразовании системы координат, мы получаем индекс, строку и столбец вокселя в центре нашего узла.

С помощью координат (I, R, C) мы можем взять небольшой 3D-срез наших данных КТ, чтобы использовать его в качестве входных данных для нашей модели. Помимо массива 3D-данных, мы должны сформировать остальную часть нашего кортежа обучающих данных, включив в него массив точек, флаг состояния узелка, UID и индекс этого образца в списке кандидатов на узелки КТ. Именно такой кортеж данных PyTorch будет ожидать от нашего подкласса `Dataset`, и он является последней частью моста от наших исходных необработанных данных до стандартной структуры тензоров PyTorch.

Важно ограничивать или обрезать данные, чтобы не «утопить» нашу модель в шуме, но в то же время не следует обрезать их, начиная от входа. Диапазон данных должен обеспечивать правильное поведение, особенно после нормализации. Полезно бывает ограничивать данные для удаления выбивающихся точек, особенно если таковые встречаются в данных от природы. Мы также можем реализовать алгоритмические преобразования входных данных вручную,

то есть заняться *конструированием признаков*, о котором мы кратко говорили в главе 1. Обычно мы хотим, чтобы модель брала на себя основную тяжелую работу. Конструирование признаков находит свое применение, но в этом проекте мы не будем прибегать к нему.

10.2. ПАРСИНГ ДАННЫХ АННОТАЦИЙ LUNA

Первым делом нам нужно начать загрузку данных. При работе над новым проектом часто начинают именно с этого. Мы должны быть уверены, что знаем, как работать с необработанным вводом, а понимание структуры данных после загрузки позволяет уверенно проводить первые эксперименты. Мы могли бы попробовать загружать отдельные КТ-сканы, но нам кажется, что лучше проанализировать файлы CSV от LUNA, которые содержат информацию о целевых точках в каждом КТ-скане. На рис. 10.3 видно, что мы ожидаем получить некую информацию о координатах, отметку о том, является ли эта координата узелком, и уникальный идентификатор КТ-скана. Поскольку в файлах CSV меньше типов информации и их легче парсить, мы надеемся, что это поможет вам лучше понять, что искать, когда мы начнем загружать сканы.

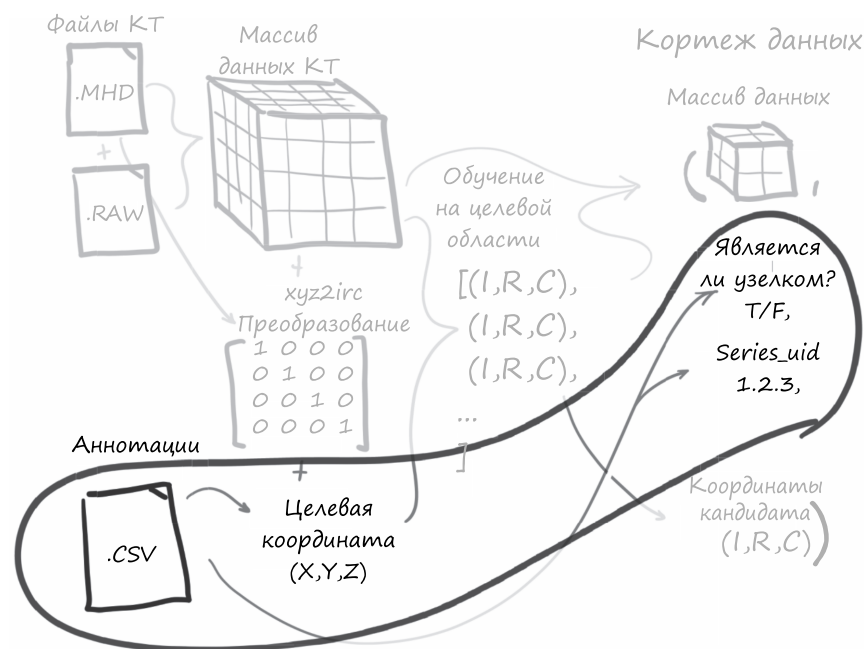


Рис. 10.3. Аннотации LUNA в файле candidates.csv содержат серию КТ, положение кандидата и флаг, указывающий, является ли кандидат узелком

Файл `candidates.csv` содержит информацию обо всех образованиях, которые потенциально выглядят как узелки, независимо от того, являются они злокачественными, доброкачественными опухолями или вообще чем-то другим. Эти данные мы будем использовать в качестве основы для создания полного списка кандидатов, который затем можно разделить на обучающие и проверочные данные. В следующем сеансе оболочки Bash показано, что содержит файл:

```
$ wc -l candidates.csv  ← Подсчет числа строк в файле
551066 candidates.csv

$ head data/part2/luna/candidates.csv  ← Вывод первых нескольких строк файла
seriesuid,coordX,coordY,coordZ,class  ← Первая строка файла .csv
1.3...6860,-56.08,-67.85,-311.92,0      содержит заголовки
1.3...6860,53.21,-244.41,-245.17,0
1.3...6860,103.66,-121.8,-286.62,0
1.3...6860,-33.66,-72.75,-308.41,0
...

$ grep ',1$' candidates.csv | wc -l  ← Подсчет числа строк, оканчивающихся
1351                                  единицей, что является признаком
                                      злокачественного узелка
```

ПРИМЕЧАНИЕ

Значения в столбце `seriesuid` были опущены в целях лучшего отображения на странице.

В итоге мы получаем 551 000 строк, у каждой из которых есть `seriesuid` (который мы назовем в коде `series_uid`), координаты (X, Y, Z) и столбец `class`, соответствующий статусу узелка (это логическое значение: 0, если кандидат не является узелком, и 1 — если является злокачественным или доброкачественным узелком). У нас есть 1351 кандидат, являющийся узелком.

Файл `annotations.csv` содержит информацию о некоторых кандидатах, помеченных как узелки. В частности, нас интересует параметр `diameter_mm`:

```
$ wc -l annotations.csv
1187 annotations.csv  ← Это не то же число, что в файле candidates.csv

$ head data/part2/luna/annotations.csv
seriesuid,coordX,coordY,coordZ,diameter_mm  ← Последний столбец тоже отличается
1.3.6...6860,-128.6994211,-175.3192718,-298.3875064,5.651470635
1.3.6...6860,103.7836509,-211.9251487,-227.12125,4.224708481
1.3.6...5208,69.63901724,-140.9445859,876.3744957,5.786347814
1.3.6...0405,-24.0138242,192.1024053,-391.0812764,8.143261683
...
```

У нас есть информация о размерах около 1200 узелков. Это полезно, поскольку она позволяет нам включить в обучающие и проверочные данные репрезентативный разброс размеров узелков. Без этого может оказаться, что в проверочные данные попадут только экстремальные значения, и будет создаваться впечатление, что модель неэффективна.

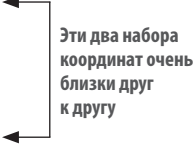
10.2.1. Обучающие и проверочные наборы

Для любой стандартной задачи обучения с учителем (типичным примером которой является классификация) данные делятся на обучающий и проверочный наборы. Оба должны быть *репрезентативны* для диапазона реальных входных данных, которые мы ожидаем получить и хотим обрабатывать. Если какой-либо из наборов существенно отличается от реальных вариантов использования, то вполне вероятно, что наша модель будет вести себя не так, как мы ожидаем, поскольку модель, обученная на далеких от реальности данных, не сможет нормально работать в полевых условиях! Мы не пытаемся вводить какие-то точные правила, но в будущих проектах вам нужно следить за тем, чтобы обучение и проверка выполнялась только на данных, похожих на реальные.

Вернемся к узелкам. Нам нужно отсортировать их по размеру и взять для проверочного набора каждый N -й. Это позволит получить репрезентативный разброс, который нам и нужен. К сожалению, информация о местоположении, указанная в файле `annotations.csv`, не всегда точно совпадает с координатами в файле `candidates.csv`:

```
$ grep 100225287222365663678666836860 annotations.csv
1.3.6...6860,-128.6994211,-175.3192718,-298.3875064,5.651470635
1.3.6...6860,103.7836509,-211.9251487,-227.12125,4.224708481

$ grep '100225287222365663678666836860.*,1$' candidates.csv
1.3.6...6860,104.16480444,-211.685591018,-227.011363746,1
1.3.6...6860,-128.94,-175.04,-297.87,1
```



Эти два набора координат очень близки друг к другу

Если мы возьмем соответствующие координаты из каждого файла, то получим $(-128,70, -175,32, -298,39)$ и $(-128,94, -175,04, -297,87)$. Рассматриваемый узелок имеет диаметр 5 мм, притом эти точки обозначают «центр» узелка, но слегка отличаются. Было бы совершенно логично решить, что работать с несоответствующими данными не стоит, и проигнорировать файл. Однако мы немного поработаем, чтобы привести все в соответствие, поскольку наборы данных реального мира часто бывают несовершенны и это хороший пример той работы, которую вам часто нужно будет проделывать в случае сбора данных из разных источников.

10.2.2. Объединение аннотаций и данных кандидатов

Когда мы знаем, как выглядят наши файлы необработанных данных, нужно создать функцию `getCandidateInfoList`, которая объединит все данные вместе. Для хранения информации о каждом узелке будем использовать именованный кортеж, определенный в верхней части файла (листинг 10.1).

Листинг 10.1. dsets.py:7

```

from collections import namedtuple
# ... строка 27
CandidateInfoTuple = namedtuple(
    'CandidateInfoTuple',
    'isNodule_bool, diameter_mm, series_uid, center_xyz',
)

```

Полученные кортежи не слишком подходят для обучения, поскольку в них отсутствуют нужные нам фрагменты данных КТ. Зато они представляют собой отполированный и унифицированный интерфейс для аннотированных человеком данных, с которыми мы работаем. Очень важно отделить работу с беспорядочными данными от обучения модели. В противном случае цикл обучения может сильно замедлиться, ведь вам придется работать с особыми случаями и другими отвлекающими факторами прямо в середине кода, который должен заниматься обучением.

СОВЕТ

Четко отделите код, отвечающий за очистку данных, от остальной части вашего проекта. Не бойтесь переписать свои данные и при необходимости сохранить их на диск.

В списке информации о кандидатах будет лежать состояние узелка (которое модель и должна научиться классифицировать), диаметр (полезно для получения хорошего разброса обучающих данных, так как у маленьких и больших узелков разные признаки), серии (для правильного расположения КТ) и центр кандидата (чтобы найти кандидата на более крупном КТ). Функция, которая создает список экземпляров `NoduleInfoTuple`, запускается с помощью декоратора кэширования в памяти, а затем выполняется получение списка файлов, присутствующих на диске (листинг 10.2).

Листинг 10.2. dsets.py:32

```

@functools.lru_cache(1)
def getCandidateInfoList(requireOnDisk_bool=True):
    mhd_list = glob.glob('data-unversioned/part2/luna/subset*/*.mhd')
    presentOnDisk_set = {os.path.split(p)[-1][:4] for p in mhd_list}

```

Функция кэширования из стандартной библиотеки

Переменная `requireOnDisk_bool` позволяет отличить серии от наборов данных, которые еще не размещены

Поскольку парсинг некоторых файлов данных может занять время, мы кэшируем результаты вызова этой функции в памяти. Это пригодится позже, поскольку в следующих главах мы будем вызывать ее чаще. Ускорение конвейера данных за счет тщательного применения кэширования в памяти или на диске может значительно увеличить скорость обучения. Используйте такие возможности, работая над своими проектами.

Ранее мы говорили, что запуск обучающей программы будет выполняться на неполном наборе обучающих данных из-за долгой загрузки и нехватки места на диске. Параметр `requireOnDisk_bool` делает именно это: мы определяем, какие UID из данных LUNA у нас есть и готовы к загрузке с диска. С помощью этой информации мы будем ограничивать количество записей, которые достаем из CSV-файлов для будущего анализа. Возможность пропустить подмножество данных через цикл обучения может пригодиться для проверки работоспособности кода. Часто результаты обучения модели при этом бывают плохими или бесполезными, но зато мы можем извлечь полезные данные с помощью логов, метрик, контрольных точек модели и аналогичных инструментов.

Получив информацию о кандидате, мы хотим объединить информацию о диаметре из `annotations.csv`. Сначала нам нужно сгруппировать наши аннотации по `series_uid`, так как это первый ключ, который мы будем использовать для перекрестной ссылки на каждую строку из двух файлов (листинг 10.3).

Листинг 10.3. `dssets.py:40, def getCandidateInfoList`

```
diameter_dict = {}
with open('data/part2/luna/annotations.csv', "r") as f:
    for row in list(csv.reader(f))[1:]:
        series_uid = row[0]
        annotationCenter_xyz = tuple([float(x) for x in row[1:4]])
        annotationDiameter_mm = float(row[4])

        diameter_dict.setdefault(series_uid, []).append(
            (annotationCenter_xyz, annotationDiameter_mm)
        )
```

Теперь создадим полный список кандидатов, используя информацию в файле `candidates.csv` (листинг 10.4).

Листинг 10.4. `dssets.py:51, def getCandidateInfoList`

```
candidateInfo_list = []
with open('data/part2/luna/candidates.csv', "r") as f:
    for row in list(csv.reader(f))[1:]:
        series_uid = row[0]

        if series_uid not in presentOnDisk_set and requireOnDisk_bool:
            continue

        isNodule_bool = bool(int(row[4]))
        candidateCenter_xyz = tuple([float(x) for x in row[1:4]])

        candidateDiameter_mm = 0.0
        for annotation_tup in diameter_dict.get(series_uid, []):
            annotationCenter_xyz, annotationDiameter_mm = annotation_tup
            for i in range(3):
                delta_mm = abs(candidateCenter_xyz[i] - annotationCenter_xyz[i])
```

Если `series_uid` отсутствует,
то соответствующих данных
у нас нет на диске, поэтому
мы должны его пропустить

```

        if delta_mm > annotationDiameter_mm / 4:
            break
        else:
            candidateDiameter_mm = annotationDiameter_mm
            break

candidateInfo_list.append(CandidateInfoTuple(
    isNodule_bool,
    candidateDiameter_mm,
    series_uid,
    candidateCenter_xyz,
))

```

Делим диаметр на 2, чтобы получить радиус, и делим радиус на 2, чтобы две центральные точки узелка не находились слишком далеко друг от друга относительно размера узелка (в результате мы проверяем ограничивающую рамку, а не настоящее расстояние)

Для всех кандидатов для данного `series_uid` мы просматриваем аннотации, которые получили ранее, и смотрим, достаточно ли близки две координаты, чтобы считать их одним и тем же узелком. Если да, то отлично! Теперь у нас есть информация о диаметре этого узелка. Если же мы не найдем совпадения, то ничего страшного, просто будем рассматривать узелок как имеющий диаметр 0,0. Мы используем эту информацию только для получения хорошего разброса размеров узелков в обучающих и проверочных выборках, и неправильные диаметры некоторых узелков не должны быть проблемой. Однако нужно помнить, что мы делаем это на случай, если наше предположение здесь неверно.

У нас будет много довольно неудобного кода, который занимается именно объединением диаметров. К сожалению, такие манипуляции и нечеткое сопоставление иногда применяются достаточно часто, в зависимости от исходных данных. Зато потом нам просто нужно отсортировать данные и вернуть их (листинг 10.5).

Листинг 10.5. `dsets.py:80, def getCandidateInfoList`

```

candidateInfo_list.sort(reverse=True)
return candidateInfo_list

```

Мы получаем все имеющиеся данные узелков, начиная с самого крупного, а затем идут данные, не содержащие узелков (без информации о размере узелка)

Порядок членов в `noduleInfo_list` определяется именно этой сортировкой. Мы используем такой подход к сортировке, чтобы в любом срезе данных иметь репрезентативную часть узелков с хорошим разбросом диаметров. Мы обсудим это подробнее в подразделе 10.5.3.

10.3. ЗАГРУЗКА СКАНОВ КТ

Теперь нам нужна возможность брать данные КТ из кучи битов на диске и превращать их в объект Python, из которого мы можем извлекать трехмерные данные о плотности узелков. Взгляните на путь от файлов `.mhd` и `.raw` к объектам `Ct` на рис. 10.4. Информация об аннотациях узелков работает как карта точек интереса

в необработанных данных. Прежде чем мы сможем по этой карте находить интересные нас данные, нужно будет преобразовать данные в адресную форму.

СОВЕТ

Наличие большого количества необработанных данных, большая часть которых не представляет интереса, — обычная ситуация. При работе над собственными проектами ищите способы ограничить область работы только релевантными данными.

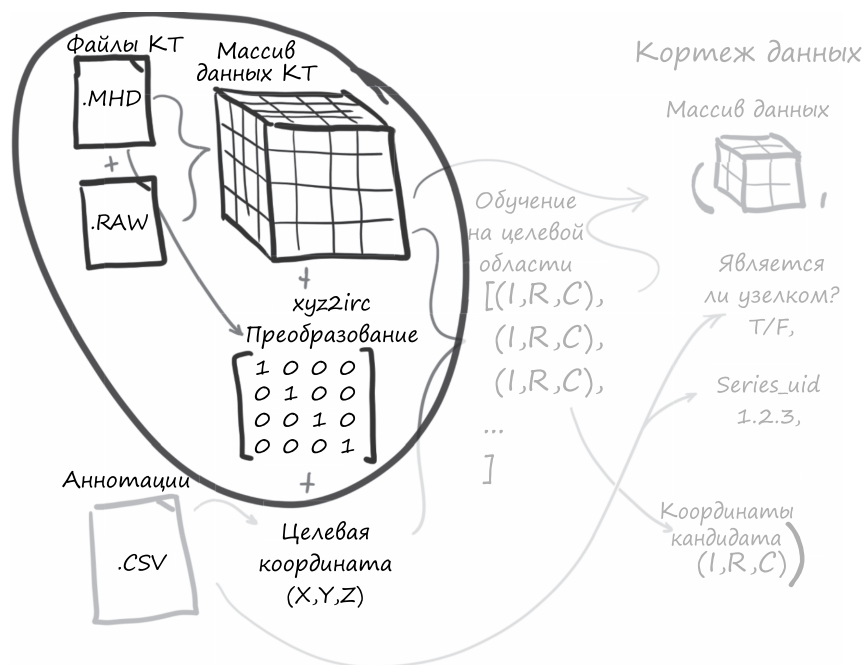


Рис. 10.4. Загрузка КТ-скана создает массив вокселей и преобразует координаты пациента в индексы массива

ПРИМЕЧАНИЕ

Мы проделали немало работы по поиску подходящей библиотеки для парсинга файлов необработанных данных, а для других форматов, о которых вы раньше не слышали, вам придется найти библиотеку самостоятельно. Рекомендуем потратить на это время! В экосистеме Python есть парсеры практически для всех известных форматов файлов, и ваше время почти наверняка лучше уделить новым частям вашего проекта, чем написанию парсеров для неизвестных форматов данных.

У файлов КТ-сканов есть собственный формат под названием DICOM (www.dicomstandard.org). Первая версия стандарта DICOM была разработана в 1984 году, и, как и следовало ожидать от любых начинаний того времени, связанных с вычислениями, формат сегодня выглядит не слишком

привлекательно (например, целые разделы, которые сейчас устарели, были посвящены используемому протоколу канального уровня, поскольку Ethernet тогда еще не вошел в моду).

К счастью, в LUNA данные, с которыми мы будем взаимодействовать в этой главе, уже преобразовали в формат MetaIO, он несколько проще в использовании (https://itk.org/Wiki/MetaIO/Documentation#Quick_Start). Не волнуйтесь, если вы никогда раньше не слышали о нем! Формат файлов данных для нас может оставаться «черным ящиком», а с помощью SimpleITK мы можем превратить их в более привычные массивы NumPy (листинг 10.6).

Листинг 10.6. `dsets.py:9`

```
import SimpleITK as sitk
# ... строка 83
class Ct:
    def __init__(self, series_uid):
        mhd_path = glob.glob(
            'data-unversioned/part2/luna/subset*/{}.mhd'.format(series_uid)
        )[0]

        ct_mhd = sitk.ReadImage(mhd_path)
        ct_a = np.array(sitk.GetArrayFromImage(ct_mhd), dtype=np.float32)
```

Нам неважно, к какому подмножеству принадлежит заданный `series_uid`, поэтому мы используем подстановочный знак

`sitk.ReadImage` неявно берет на вход `.raw`-файл вместе с переданным файлом `.mhd`

Воссоздает `np.array`, так как мы хотим преобразовать тип значения в `np.float32`

В реальных проектах приходится анализировать, какие типы информации содержатся в необработанных данных, но вполне в порядке вещей для анализа битов на диске использовать и сторонний код, такой как SimpleITK. Чтобы найти золотую середину между знанием всего о входных данных и слепым принятием того, что вам дает библиотека, выбранная для работы с ними, нужен некоторый опыт. Просто помните: нас в основном интересуют *данные*, а не *биты*. Важна сама информация, а не то, как она представлена.

Возможность однозначно идентифицировать некую выборку данных бывает полезна. Например, информация о том, какая именно выборка данных вызывает проблему или дает плохие результаты классификации, может значительно помочь нам в поиске причины проблемы и ее устранении. В зависимости от характера данных уникальный идентификатор может быть атомарным, допустим числом или строкой, а может быть и более сложным, например кортежем.

Мы идентифицируем КТ-сканы с помощью *UID экземпляра серии* (`series_uid`), который присваивается при создании КТ-скана. В DICOM уникальные идентификаторы (UID) активно используются для отдельных файлов DICOM, групп файлов, курсов лечения и т. д. Эти идентификаторы аналогичны концепции UUID (<https://docs.python.org/3.6/library/uuid.html>), но создаются и форматируются по-другому. Для наших целей мы можем рассматривать их как непрозрачные

строки ASCII, выполняющие роль уникальных ключей КТ-сканов. Официально в UID DICOM могут использоваться только символы от 0 до 9 и точки (.), но некоторые файлы DICOM в реальных проектах преобразуются так, чтобы заменить UID шестнадцатеричными (0–9 и a–f) или другими не соответствующими спецификации значениями (при этом значения, не соответствующие спецификации, обычно не помечаются и не очищаются синтаксическими анализаторами DICOM, что в итоге создает беспорядок).

В десяти подмножествах, о которых мы упоминали ранее, содержится примерно по 90 КТ-сканов (всего их 888), причем каждый из них представлен в виде двух файлов: одного с расширением `.mhd` и одного с расширением `.raw`. Данные, распределенные по нескольким файлам, обрабатываются в процедуре `sitk`, но нам об этом задумываться не стоит.

На данный момент `ct_a` представляет собой трехмерный массив. Все три измерения являются пространственными, а канал интенсивности задан неявно. Как мы видели в главе 4, в тензоре PyTorch информация о канале представлена в виде четвертого измерения размером 1.

10.3.1. Единицы Хаунсфилда

Ранее мы говорили, что нам нужно понимать суть *данных*, а не копаться в кодирующих их *битах*. У нас есть прекрасный пример, чтобы продемонстрировать это. Не понимая нюансов значений и возможных диапазонов данных, мы невольно «накормим» модель данными, на которых она не сможет правильно обучиться.

Продолжим писать метод `__init__`. Теперь нам нужно немного подчистить значения в `ct_a`. Воксели КТ выражены в единицах Хаунсфилда (HU, подробнее тут: https://en.wikipedia.org/wiki/Hounsfield_scale), где воздух имеет значение -1000 HU (для нас это достаточно близко к 0 г/см³), вода составляет 0 HU (1 г/см³), а кость — не менее $+1000$ HU ($2\text{--}3$ г/см³).

ПРИМЕЧАНИЕ

Значения HU обычно хранятся на диске в виде 12-битных целых чисел со знаком (вставленных в 16-битные целые числа), что хорошо соответствует уровню точности, который обеспечивают компьютерные томографы. Это довольно интересно, но не имеет особого отношения к проекту.

Некоторые компьютерные томографы используют значения HU, соответствующие отрицательной плотности, указывая с их помощью на воксели, находящиеся за пределами поля зрения компьютерного томографа. Для наших целей все, что находится за пределами пациента, должно считаться воздухом, поэтому мы

отбрасываем информацию о поле зрения, устанавливая нижнюю границу значений на уровне -1000 HU. Аналогично точная плотность костей, металлических имплантатов и так далее нас тоже не интересует, так что мы ограничиваем ее значением примерно 2 г/см^3 (1000 HU), хотя в большинстве случаев это биологически неточно (листинг 10.7).

Листинг 10.7. dsets.py:96, Ct.__init__
`ct_a.clip(-1000, 1000, ct_a)`

Значения выше 0 HU не идеально соотносятся с плотностью, но интересующие нас опухоли обычно имеют плотность около 1 г/см^3 (0 HU), поэтому мы будем игнорировать тот факт, что HU не идеально соотносятся с общепринятыми единицами наподобие г/см^3 . Это не страшно, так как наша модель будет учиться работать с HU напрямую.

Нам следует удалить из наших данных все выбивающиеся значения. Они не нужны для нашей цели, а их наличие может усложнить работу модели. Это усложнение может произойти по-разному, но чаще всего возникает ситуация, когда при пакетной нормализации выбивающиеся значения искажают данные. Всегда ищите способы удалить из данных все лишнее.

Все созданные нами ценности теперь присваиваются `self` (листинг 10.8).

Листинг 10.8. dsets.py:98, Ct.__init__
`self.series_uid = series_uid`
`self.hu_a = ct_a`

Важно помнить, что наши данные лежат в диапазоне от -1000 до $+1000$, и в конце главы 13 мы добавим к данным каналы информации. Если не учесть несоответствие между HU и дополнительными данными, то новые каналы могут исказиться из-за влияния необработанных значений HU. На этапе классификации мы не будем добавлять больше каналов данных, так что прямо сейчас можно не заниматься этим вопросом.

10.4. ОПРЕДЕЛЕНИЕ ПОЛОЖЕНИЯ УЗЕЛКА В СИСТЕМЕ КООРДИНАТ ПАЦИЕНТА

Из-за наличия фиксированного количества входных нейронов моделям глубокого обучения обычно требуются входные данные фиксированного размера¹. Нам нужно создать массив фиксированного размера, в котором содержится

¹ Исключения есть, но они сейчас не актуальны.

кандидат. Такой массив можно задействовать в качестве входных данных для нашего классификатора. Для обучения модели мы хотим использовать кадр компьютерной томографии, на котором кандидат расположен по центру, поскольку в этом случае модели не придется учиться находить узелки, спрятанные в углу картинки. Уменьшая вариацию ожидаемых входных данных, мы упрощаем работу модели.

10.4.1. Система координат пациента

К сожалению, все данные центра кандидата, которые мы загрузили в разделе 10.2, выражены в миллиметрах, а не вокселях! Мы не можем просто подставить расстояние в миллиметрах в индекс массива и ожидать, что все заработает как надо. Как видно из рис. 10.5, нам нужно преобразовать наши координаты из миллиметровой системы координат (X, Y, Z), в которой они выражены, в систему координат, основанную на адресах вокселей (I, R, C), используемую для получения срезов массива из данных компьютерной томографии. Это классический пример того, как важно правильно работать с единицами измерения!

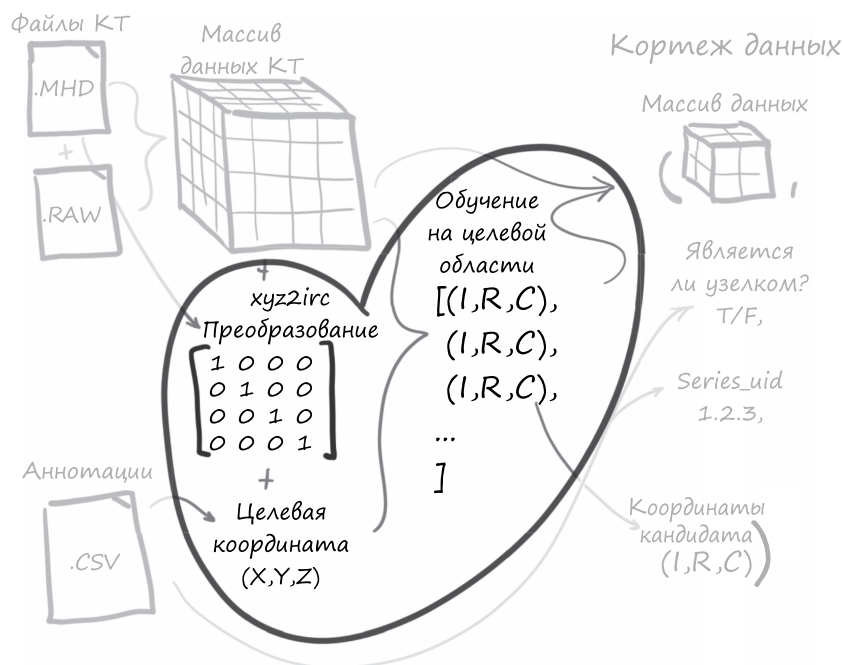


Рис. 10.5. Преобразование координат центра узла в координатах пациента (X, Y, Z) в индексы массива (индекс, строка, столбец)

Как мы упоминали ранее, при работе с компьютерной томографией мы называем размерности массива *индексом*, *строкой* и *столбцом*, поскольку для X , Y и Z существует отдельное значение, как показано на рис. 10.6. В *системе координат пациента* положительное значение X определяется как направление к левой стороне тела пациента (*влево*), положительное значение Y — как направление к спине (*назад*) и положительное значение Z — как направление к голове пациента (*вверх*). Эту систему иногда сокращенно называют LPS — left-posterior-superior, или «влево — назад — вверх».

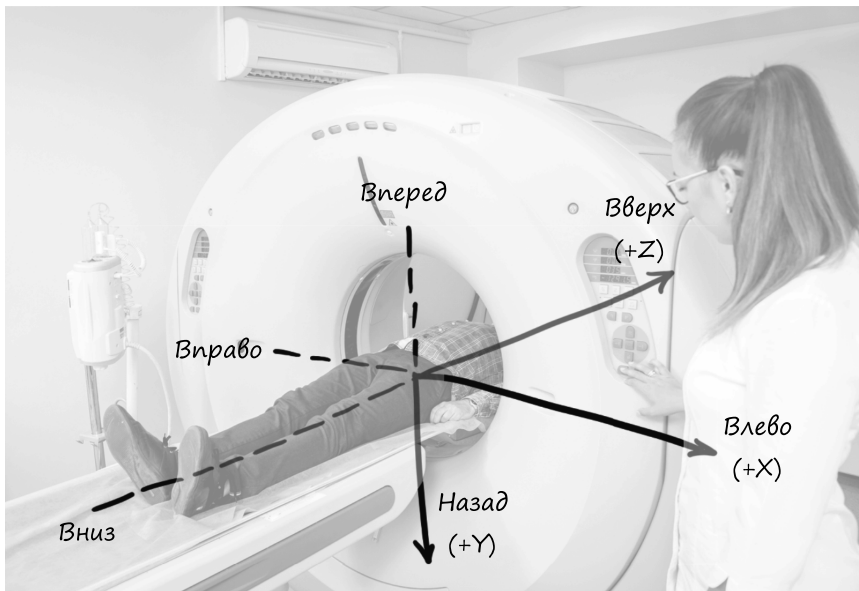


Рис. 10.6. Наш неподобающе одетый пациент и его оси координат

Система координат пациента измеряется в миллиметрах, а начало координат в ней расположено произвольно и не соответствует началу координат массива вокселей, как показано на рис. 10.7.

Система координат пациента часто используется для указания местоположения анатомически интересных мест независимо от сканера. Метаданные, определяющие связь между массивом КТ и системой координат пациента, хранятся в заголовке файлов DICOM. Этот формат метаизображений сохраняет данные в своем заголовке. Метаданные о связи систем координат позволяют нам построить преобразование из (X, Y, Z) в (I, R, C) , которое мы видели на рис. 10.5. В необработанных данных содержится много других полей с похожими метаданными, но, поскольку они нам сейчас не нужны, мы их проигнорируем.

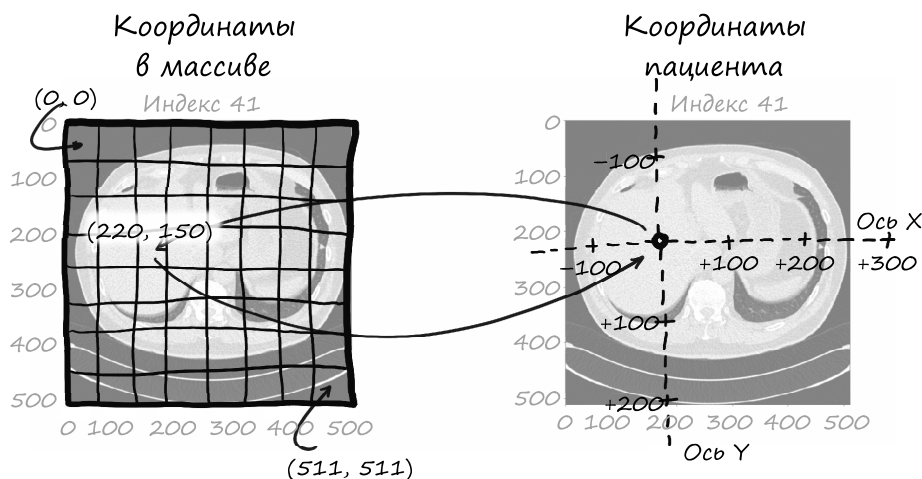


Рис. 10.7. Координаты массива и координаты пациента различаются началом отсчета и масштабом

10.4.2. Форма КТ-скана и размеры вокселя

Многие компьютерные томографы часто отличаются друг от друга размером вокселей, которые обычно не кубической формы. Они могут иметь размеры, например, $1,125 \times 1,125 \times 2,5$ мм. Обычно размеры строк и столбцов одинаковы, а размер индекса несколько больше, но могут использоваться и другие соотношения.

При построении изображения с использованием квадратных пикселей некубические воксели могут исказиться, подобно тому как искажается карта мира вблизи Северного и Южного полюсов в проекции Меркатора. Но это не самая точная аналогия, поскольку в случае с КТ искажение однородно и линейно, а именно фигура пациента кажется более приземистой или бочкообразной, как на рис. 10.8. Если мы хотим, чтобы изображения показывали реалистичные пропорции, то нужно будет применить коэффициент масштабирования.

Знание подобных подробностей значительно помогает визуальнo интерпретировать полученные результаты. Не зная особенностей, мы бы подумали, что в процессе загрузки что-то пошло не так, например, мы пропустили половину вокселей и пациент на картинке из-за этого визуальнo сжался. В результате вы можете легко потратить много времени на исправление ошибки, которой не существует. А чтобы этого не случилось, нужно понимать природу данных.

КТ-сканы обычно имеют размерность 512 строк на 512 столбцов, а по оси индексов обычно бывает от 100 полных срезов до, возможно, 250 срезов (250 срезов по 2,5 мм обычно достаточно, чтобы охватить интересующую анатомическую

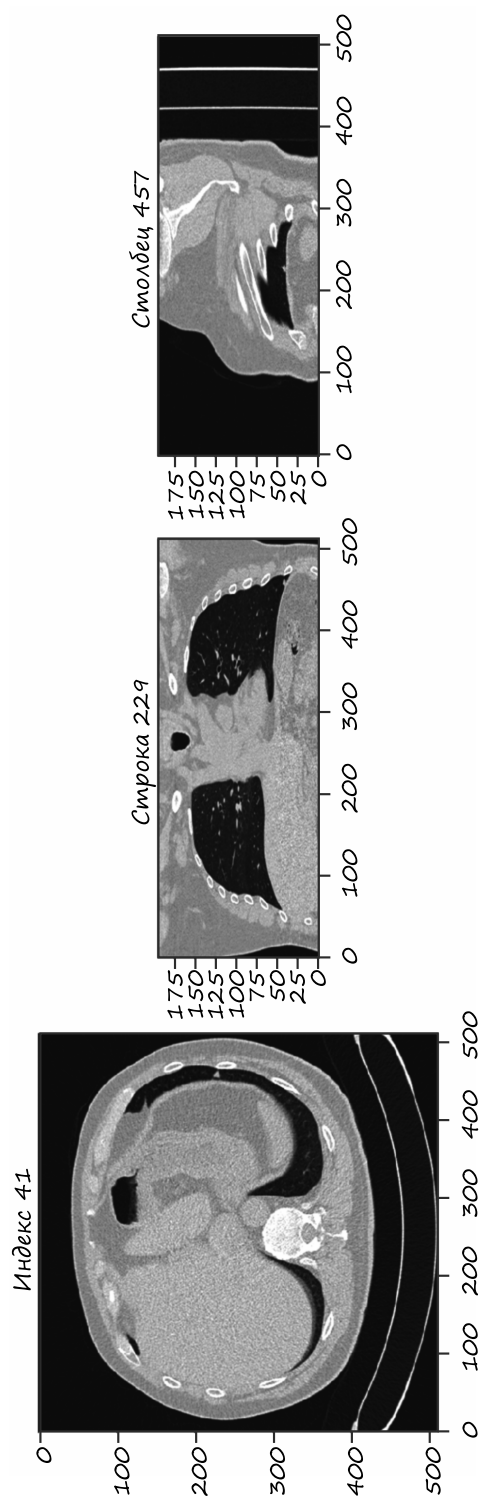


Рис. 10.8. Компьютерная томография с некубическими вокселями вдоль оси индексов. Обратите внимание, что легкие сжаты в вертикальном направлении

область). В результате мы получаем нижнюю границу 2^{25} вокселей, или около 32 миллионов точек данных. В метаданных файла каждого скана указан размер вокселя в миллиметрах, который мы получим методом `ct_mhd.GetSpacing()` в листинге 10.10.

10.4.3. Преобразование миллиметров в адреса вокселей

Напишем вспомогательный код, который будет выполнять преобразование из координат пациента в миллиметрах (их мы будем обозначать в коде суффиксами `_xyz` в именах переменных) в координаты массива (I,R,C) (их мы будем обозначать в коде суффиксами `_irc`).

Вы можете спросить, нет ли в библиотеке SimpleITK готовых функций для такого преобразования. И действительно, у экземпляра `Image` есть два метода: `TransformIndexToPhysicalPoint` и `TransformPhysicalPointToIndex`, которые именно это и делают (за исключением отражения из CRI [столбец, строка, индекс] в IRC). Однако нам нужна возможность выполнять эти вычисления, не сохраняя объект `Image`, поэтому мы выполним математические операции вручную.

Переворот осей (и, возможно, вращение или другие преобразования) кодируется в матрице 3×3 , которая возвращается в виде кортежа методом `ct_mhd.GetDirections()`. Чтобы перейти от индексов вокселей к координатам, нам нужно выполнить следующие четыре шага по порядку.

1. Преобразовать координаты из IRC в CRI, чтобы привести их в соответствие с XYZ.
2. Масштабировать индексы под размеры вокселя.
3. Выполнить матричное умножение с матрицей направлений с помощью оператора `@` в Python.
4. Добавить смещение начала координат.

Чтобы перейти от XYZ к IRC, нужно выполнить инверсию каждого шага в обратном порядке.

Размеры вокселей сохраняются в именованных кортежах, поэтому мы преобразуем их в массивы (листинг 10.9).

Да уж. Если задачка показалась вам тяжелой, то не волнуйтесь. Просто помните, что нам нужно выполнить преобразование, используя функции как «черный ящик». Метаданные, которые нам нужно преобразовать из координат пациента (`_xyz`) в координаты массива (`_irc`), содержатся в файле `MetaIO` вместе с самими данными КТ. Мы извлекаем метаданные о размере и положении вокселей из файла `.mhd` одновременно с получением `ct_a` (листинг 10.10).

Листинг 10.9. util.py:16

Меняем порядок при преобразовании
в массив NumPy

```
IrcTuple = collections.namedtuple('IrcTuple', ['index', 'row', 'col'])
XyzTuple = collections.namedtuple('XyzTuple', ['x', 'y', 'z'])
```

```
def irc2xyz(coord_irc, origin_xyz, vxSize_xyz, direction_a):
```

```
→ cri_a = np.array(coord_irc)[::-1]
```

```
origin_a = np.array(origin_xyz)
```

```
vxSize_a = np.array(vxSize_xyz)
```

```
coords_xyz = (direction_a @ (cri_a * vxSize_a)) + origin_a
```

```
return XyzTuple(*coords_xyz)
```

Три последних шага
выполняются в одну строку

```
def xyz2irc(coord_xyz, origin_xyz, vxSize_xyz, direction_a):
```

```
origin_a = np.array(origin_xyz)
```

```
vxSize_a = np.array(vxSize_xyz)
```

```
coord_a = np.array(coord_xyz)
```

```
cri_a = ((coord_a - origin_a) @ np.linalg.inv(direction_a)) / vxSize_a
```

Инверсия последних трех шагов

```
→ cri_a = np.round(cri_a)
```

```
return IrcTuple(int(cri_a[2]), int(cri_a[1]), int(cri_a[0]))
```

Аккуратное округление перед
преобразованием в целые числа

Изменение порядка
и преобразование в целые числа

Листинг 10.10. dsets.py:72, класс Ct

```
class Ct:
```

```
def __init__(self, series_uid):
```

```
    mhd_path = glob.glob('data-
```

```
    unversioned/part2/luna/subset*/{ }.mhd'.format(series_uid))[0]
```

```
    ct_mhd = sitk.ReadImage(mhd_path)
```

```
    # ... строка 91
```

```
    self.origin_xyz = XyzTuple(*ct_mhd.GetOrigin())
```

```
    self.vxSize_xyz = XyzTuple(*ct_mhd.GetSpacing())
```

```
    self.direction_a = np.array(ct_mhd.GetDirection()).reshape(3, 3)
```

Преобразуем направления массива
и превращаем массив из девяти элементов
в матрицу формы 3 × 3

Это входные данные, которые нам нужно передать в функцию преобразования `xyz2irc` вместе с самими точками. Теперь у объекта КТ есть все данные, необходимые для преобразования центра кандидата из координат пациента в координаты массива.

10.4.4. Извлечение узелка из скана КТ

Как мы упоминали в главе 9, в 999 999 % вокселей на КТ пациента узелков не будет (а значит, и рака, если уж на то пошло). Подобное соотношение может быть эквивалентно двухпиксельному пятну неправильного цвета в телевизионной передаче в высоком разрешении или одному написанному с ошибкой слову на целой книжной полке. Заставить модель исследовать такие огромные

массивы данных в поисках намеков на нужные нам узелки — это все равно что попросить вас найти одно слово с ошибкой в сборнике романов, написанных на незнакомом вам языке!

Вместо этого, как показано на рис. 10.9, мы выделим область вокруг каждого кандидата и заставим модель рассматривать кандидатов по одному. В примере с романами это было бы аналогично чтению книг по одному абзацу за раз — это все еще непросто, но уже не настолько. В целом поиск возможностей способов облегчить модели работу весьма полезен, особенно на ранних стадиях проекта, когда мы пытаемся запустить нашу первую рабочую реализацию.

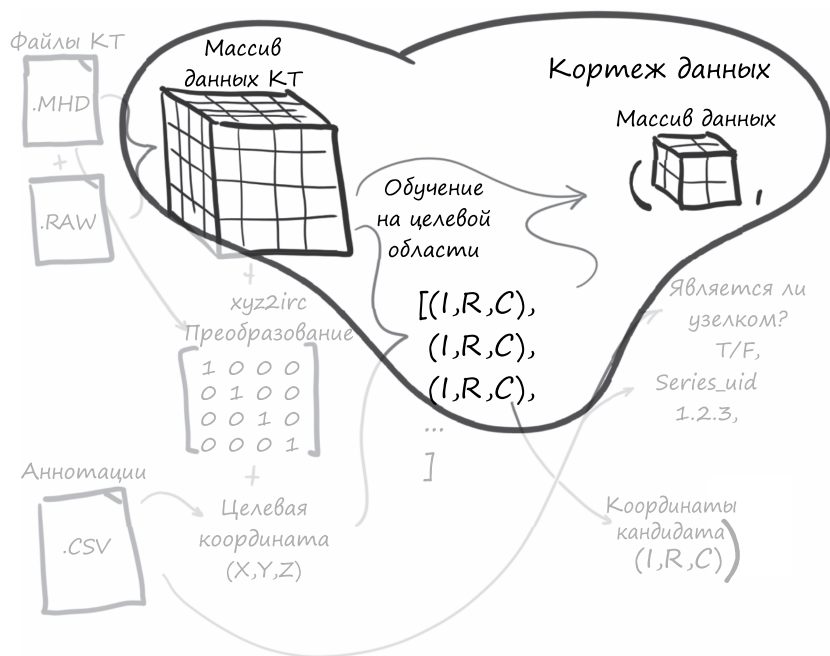


Рис. 10.9. Взятие области вокруг точки-кандидата из большого массива вокселей КТ с использованием информации о координатах его центра (индекс, строка, столбец)

Функция `getRowNodule` принимает центр кандидата в системе координат пациента (X, Y, Z) из CSV-данных LUNA, а также ширину в вокселях. Возвращает кубический фрагмент КТ, а также центр кандидата, преобразованный в систему массива (листинг 10.11).

Листинг 10.11. `dsets.py:105, Ct.getRowCandidate`

```
def getRowCandidate(self, center_xyz, width_irc):
    center_irc = xyz2irc(
        center_xyz,
```



```

        self.origin_xyz,
        self.vxSize_xyz,
        self.direction_a,
    )

    slice_list = []
    for axis, center_val in enumerate(center_irc):
        start_ndx = int(round(center_val - width_irc[axis]/2))
        end_ndx = int(start_ndx + width_irc[axis])
        slice_list.append(slice(start_ndx, end_ndx))

    ct_chunk = self.hu_a[tuple(slice_list)]

    return ct_chunk, center_irc

```

В фактической реализации нужно было бы обрабатывать ситуации, когда комбинация `center` и `width` оказывается такой, что край области выходит за пределы массива. Но, как отмечалось ранее, мы пропустим усложнения, которые будут мешать понять общее назначение функции. Полную реализацию можно найти на сайте книги (www.manning.com/books/deep-learning-with-pytorch?query=pytorch) и в репозитории GitHub (<https://github.com/deep-learning-with-pytorch/dlwpt-code>).

10.5. ПРОСТАЯ РЕАЛИЗАЦИЯ DATASET

Мы уже видели экземпляры `Dataset` PyTorch в главе 7, но здесь мы впервые сами их реализуем. Создав подкласс `Dataset`, мы возьмем наши произвольные данные и подключим их к остальной части экосистемы PyTorch. Каждый экземпляр `Ct` содержит сотни точек данных, с помощью которых мы можем обучать нашу модель или проверять ее эффективность. Наш класс `LunaDataset` нормализует эти образцы, объединяя узелки каждого КТ в единую коллекцию, из которой можно извлечь точки данных, независимо от того, из какого экземпляра КТ они взяты. Подобные техники выравнивания часто используются для обработки данных, хотя, как мы увидим в главе 12, иногда простого выравнивания данных недостаточно для хорошего обучения модели.

Что касается реализации, то мы начнем с требований, предъявляемых к подклассу `Dataset`, а затем будем двигаться в обратном направлении. Полученный результат будет отличаться от наборов данных, с которыми мы работали ранее, — там мы задействовали классы, предоставляемые внешними библиотеками, а здесь нам нужно реализовать и создать экземпляр класса самостоятельно. Но после реализации мы будем использовать его так же, как и в предыдущих примерах. К счастью, реализация пользовательского подкласса не будет слишком сложной, так как PyTorch API требует, чтобы любые подклассы `Dataset`, которые мы хотим реализовать, реализовывали лишь две функции:

- реализация `__len__`, которая после инициализации должна возвращать постоянное значение (в ряде случаев это значение кэшируется);

- метод `__getitem__`, принимающий индекс и возвращающий кортеж с демонстрационными данными, которые будут использоваться для обучения (или проверки, в зависимости от обстоятельств).

Для начала посмотрим, как выглядят сигнатуры функций и возвращаемые значения этих функций (листинг 10.12).

Листинг 10.12. `dsets.py:176, LunaDataset.__len__`

```
def __len__(self):
    return len(self.candidateInfo_list)

def __getitem__(self, ndx):
    # ... строка 200
    return (
        candidate_t, 1((C010-1))
        pos_t, 1((C010-2))
        candidateInfo_tup.series_uid, | Обучающая выборка
        torch.tensor(center_irc),
    )
```

Наша реализация `__len__` проста: у нас есть список кандидатов, где каждый кандидат — точка данных, а наш набор данных имеет размер, равный количеству этих точек. Не обязательно делать реализацию настолько простой, и в последующих главах мы изменим кое-что!¹ Единственное правило состоит в том, что если метод `__len__` возвращает значение N , то метод `__getitem__` должен возвращать что-то действительное для всех входов от 0 до $N - 1$.

Для метода `__getitem__` мы берем `ndx` (обычно целое число, учитывая правило поддержки входных данных от 0 до $N - 1$) и возвращаем образец кортежа из четырех элементов, как показано на рис. 10.2. Однако построить этот кортеж немного сложнее, чем получить длину нашего набора данных, так что взглянем более пристально. Первая часть этого метода подразумевает, что нам нужно построить `self.candidateInfo_list` и предоставить функцию `getCtRawNodule` (листинг 10.13).

Листинг 10.13. `dsets.py:179, LunaDataset.__getitem__`

```
def __getitem__(self, ndx):
    candidateInfo_tup = self.candidateInfo_list[ndx]
    width_irc = (32, 48, 48)

    candidate_a, center_irc = getCtRawCandidate(
        candidateInfo_tup.series_uid,
        candidateInfo_tup.center_xyz,
        width_irc,
    )
```

← Возвращаемое значение `candidate_a` имеет вид (32, 48, 48), где оси — глубина, высота и ширина

Подробнее поговорим в подразделах 10.5.1 и 10.5.2.

¹ На самом деле станет еще проще, но главное здесь то, что есть варианты.

Следующее, что нам нужно сделать в методе `__getitem__`, — это преобразовать данные в правильные типы данных, чтобы размерности массивов оказались подходящими для последующего кода (листинг 10.14).

Листинг 10.14. `dsets.py:189, LunaDataset.__getitem__`

```
candidate_t = torch.from_numpy(candidate_a)
candidate_t = candidate_t.to(torch.float32)
candidate_t = candidate_t.unsqueeze(0)  ← Метод .unsqueeze(0) добавляет канал 'Channel'
```

Пока не слишком беспокойтесь о том, почему мы манипулируем размерностями. В следующей главе будет содержаться код, который берет выходные данные и накладывает уже упоминаемые нами ограничения. Подобное должно быть в любом пользовательском наборе данных, который вы реализуете. Эти преобразования — важнейшая часть превращения необработанных данных в красивые упорядоченные тензоры.

Наконец, нужно построить тензор классификации (листинг 10.15).

Листинг 10.15. `dsets.py:193, LunaDataset.__getitem__`

```
pos_t = torch.tensor([
    not candidateInfo_tup.isNodule_bool,
    candidateInfo_tup.isNodule_bool
],
dtype=torch.long,
)
```

В тензоре у нас два элемента, по одному для наших возможных классов-кандидатов (узелки и не узелки, положительные или отрицательные). Мы могли бы выводить только статус узелка, но `nn.CrossEntropyLoss` ожидает одно выходное значение для каждого класса, поэтому так мы и поступим. Подробности создаваемых вами тензоров будут меняться в зависимости от типа проекта, над которым вы работаете.

Посмотрим на окончательный образец кортежа (большой вывод `nodule_t` не слишком удобочитаем, поэтому в листинге 10.16 мы опускаем его значительную часть).

Листинг 10.16. `p2ch10_explore_data.ipynb`

```
# In[10]:
LunaDataset()[0]

# Out[10]:
(tensor([[[[-899., -903., -825., ..., -901., -898., -893.],
          ...,
          [-92., -63., 4., ..., 63., 70., 52.]]]],
 tensor([0, 1])), ← ds_t
 '1.3.6...287966244644280690737019247886', ← candidate_tup.series_uid (elided)
 tensor([ 91, 360, 341])) ← center_irc
```

Здесь мы видим, что `__getitem__` возвращает четыре элемента.

10.5.1. Кэширование массивов-кандидатов с помощью функции `getCtRawCandidate`

Чтобы `LunaDataset` работал достаточно эффективно, нам нужно приложить немало усилий для кэширования на диске. Это позволит нам избежать повторного чтения всего КТ-скана для каждой точки. Это было бы ужасно расточительно! Важно уделять достаточно внимания узким местам в проекте и делать все возможное для их оптимизации, как только они начинают замедлять общую работу. Здесь мы слегка поторопились, так как не продемонстрировали, что здесь вообще нужно кэширование. Без него `LunaDataset` работает в 50 раз медленнее! Мы вернемся к этому в упражнениях в конце главы.

Сама функция проста. Это механизм кэширования (<https://pypi.python.org/pypi/diskcache>), обернутый вокруг метода `Ct.getRawCandidate`, который мы видели ранее (листинг 10.17).

Листинг 10.17. `dsets.py:139`

```
@functools.lru_cache(1, typed=True)
def getCt(series_uid):
    return Ct(series_uid)

@raw_cache.memoize(typed=True)
def getCtRawCandidate(series_uid, center_xyz, width_irc):
    ct = getCt(series_uid)
    ct_chunk, center_irc = ct.getRawCandidate(center_xyz, width_irc)
    return ct_chunk, center_irc
```

Здесь мы используем несколько различных методов кэширования. Прежде всего мы кэшируем возвращаемое значение `getCt` в памяти, чтобы можно было многократно запрашивать один и тот же экземпляр `Ct`, не загружая заново все данные с диска. Это даст огромный прирост скорости в случае повторяющихся запросов, но мы сохраняем в памяти только один КТ, поэтому промахи кэша будут частыми, если мы не будем следить за порядком доступа.

Однако функция `getCtRawCandidate`, которая вызывает `getCt`, *также* кэширует свои выходные данные; поэтому после того, как наш кэш будет заполнен, функция `getCt` вызываться не будет. Эти значения кэшируются на диск с помощью библиотеки Python `diskcache`.

Почему выбрана именно такая конфигурация кэширования, мы обсудим в главе 11. Пока достаточно знать, что считывать с диска 2^{15} значения типа `float32` намного, намного быстрее, чем читать 2^{25} значений типа `int16`, преобразовывать их в `float32`, а затем выбирать из них 2^{15} значений. Начиная со второго прохода данных, время ввода-вывода для ввода должно сократиться до незначительного значения.

ПРИМЕЧАНИЕ

Если определения функций когда-либо существенно изменятся, то нам нужно будет удалить кэшированные значения с диска. Если мы этого не сделаем, то кэш продолжит возвращать их, даже если теперь функция не будет сопоставлять входные данные со старыми выходными данными. Данные хранятся в каталоге `data-unversioned/cache`.

10.5.2. Построение набора данных в `LunaDataset.__init__`

Почти в каждом проекте вам необходимо будет разделить данные на обучающий и проверочный наборы. Сделаем это, поместив каждый десятый элемент данных, указанный параметром `val_stride`, в проверочный набор. Мы также примем параметр `isValSet_bool` и с его помощью определим, какие данные мы должны хранить: только обучающие, проверочные или все (листинг 10.18).

Листинг 10.18. `dsets.py:149`, класс `LunaDataset`

```
class LunaDataset(Dataset):
    def __init__(self,
                  val_stride=0,
                  isValSet_bool=None,
                  series_uid=None,
    ):
        self.candidateInfo_list = copy.copy(getCandidateInfoList())
        if series_uid:
            self.candidateInfo_list = [
                x for x in self.candidateInfo_list if x.series_uid == series_uid
            ]
```

Мы копируем возвращаемое значение, поэтому кэшированная копия не будет затронута изменением `self.candidateInfo_list`

Если мы передадим существующий `series_uid`, то в экземпляр попадут узелки только из указанной серии. Это может быть полезно для визуализации или отладки, а также облегчает просмотр, например, одного проблематичного скана КТ.

10.5.3. Разделение данных на обучающие и проверочные

Мы позволяем `Dataset` взять $1/N$ -ю часть данных в подмножество, используемое для проверки модели. То, как мы будем его обрабатывать, зависит от аргумента `isValSet_bool` (листинг 10.19).

Листинг 10.19. `dsets.py:162`, класс `LunaDataset`

```
if isValSet_bool:
    assert val_stride > 0, val_stride
    self.candidateInfo_list = self.candidateInfo_list[::val_stride]
    assert self.candidateInfo_list
elif val_stride > 0:
    del self.candidateInfo_list[::val_stride]
    assert self.candidateInfo_list
```

Удаление проверочных изображений (каждого элемента `val_stride` в списке) из `self.candidateInfo_list`. Мы сделали копию ранее, чтобы не изменять исходный список

Это означает, что мы можем создать два экземпляра `Dataset` и быть уверенными в том, что обучающие и проверочные данные у нас четко разделены. Конечно, это зависит от правильного порядка сортировки в `self.candidateInfo_list`, который мы обеспечиваем наличием порядка сортировки для кортежей информации-кандидата и функцией `getCandidateInfoList`, сортирующей список перед его возвратом.

Еще один момент, касающийся разделения данных на обучающие и проверочные, заключается в том, что в некоторых задачах нужно проверять, что данные от одного пациента используются либо при обучении, либо при тестировании, но не в обоих случаях. Здесь это не проблема; в противном случае нам пришлось бы разделить список пациентов и КТ-сканы, прежде чем перейти к уровню узлов.

Посмотрим на данные, используя файл `p2ch10_explore_data.ipynb`:

```
# In[2]:
from p2ch10.dsets import getCandidateInfoList, getCt, LunaDataset
candidateInfo_list = getCandidateInfoList(requireOnDisk_bool=False)
positiveInfo_list = [x for x in candidateInfo_list if x[0]]
diameter_list = [x[1] for x in positiveInfo_list]
```

```
# In[4]:
for i in range(0, len(diameter_list), 100):
    print('{:4}  {:4.1f} mm'.format(i, diameter_list[i]))
```

```
# Out[4]:
 0  32.3 mm
100 17.7 mm
200 13.0 mm
300 10.0 mm
400  8.2 mm
500  7.0 mm
600  6.3 mm
700  5.7 mm
800  5.1 mm
900  4.7 mm
1000 4.0 mm
1100 0.0 mm
1200 0.0 mm
1300 0.0 mm
```

У нас есть несколько очень крупных кандидатов, начиная с 32 мм, но затем их размер быстро уменьшается вдвое. Большинство кандидатов находятся в диапазоне от 4 до 10 мм, а у нескольких сотен вообще нет информации о размере. Это выглядит так, как и ожидалось: вы, возможно, помните, что узлов у нас было больше, чем аннотаций с указанием диаметра. Быстрая проверка ваших данных на пригодность может быть очень полезной, так как обнаружение проблемы или опровержение ошибочного предположения на ранней стадии может сэкономить часы работы!

Еще более важный вывод — чтобы хорошо работать, наши данные для обучения и проверки должны иметь несколько свойств:

- в оба набора должны включаться все разнообразные варианты ожидаемых входных данных;
- ни в одном наборе не должно быть данных, которые не являются репрезентативными для ожидаемых входных данных, если у набора нет на то специальной цели, например обучение определению каких-нибудь крайних случаев;
- обучающий набор не должен намеренно подстраивать обучение под проверочный набор, если это расходится с реальными данными (например, включение одного и того же элемента данных в оба набора или *утечка* в обучающем наборе).

10.5.4. Отображение данных

Опять же либо используйте файл `p2ch10_explore_data.ipynb` напрямую, либо запустите Jupyter Notebook и введите код:

```
# In[7]:
%matplotlib inline
from p2ch10.vis import findNoduleSamples, showNodule
noduleSample_list = findNoduleSamples()
```

Эта магическая строка настраивает возможность отображения изображений в записной книжке

СОВЕТ

Для получения дополнительной информации о том, как работает встроенная в Jupyter магия¹ matplotlib, см. <http://mng.bz/rmD>.

```
# In[8]:
series_uid = positiveSample_list[11][2]
showCandidate(series_uid)
```

Вы получите изображения, подобные КТ и срезам узелков, показанным ранее в данной главе.

Если вас это заинтересовало, то мы предлагаем вам отредактировать реализацию кода рендеринга в `p2ch10/vis.py` в соответствии с вашими потребностями и вкусами. В коде рендеринга активно используется Matplotlib (<https://matplotlib.org>), но для нас это сложная библиотека и мы не будем рассматривать ее.

Помните: рендеринг данных — не просто вывод красивых картинок. Наша цель — получить интуитивное представление о том, как выглядят ваши входные данные. Возможность с первого взгляда сказать, что «этот образец очень

¹ Так называли разработчики, не мы!

зашумлен по сравнению с остальными моими данными» или «здесь все выглядит вполне нормально», может оказаться полезной при исследовании проблем. Эффективный рендеринг также способствует генерации новых идей: «Вот *так-то и так-то* я мог бы улучшить работу программы». Подобный уровень ознакомления будет необходим, когда вы начнете браться за все более и более сложные проекты.

ПРИМЕЧАНИЕ

Из-за способа разделения каждого подмножества и сортировки, применяемой при построении `LunaDataset.candidateInfo_list`, порядок записей в `nodeSample_list` очень зависит от того, какие подмножества используются во время выполнения кода. Пожалуйста, помните об этом, когда пытаетесь найти конкретный образец во второй раз, особенно после распаковки большого количества подмножеств.

10.6. ИТОГИ ГЛАВЫ

В главе 9 мы размышляли о природе данных. А в этой главе мы заставили PyTorch поработать с ними! Преобразовав необработанные данные DICOM через мета-изображения в тензоры, мы подготовили почву для начала реализации модели и цикла обучения, которые увидим в следующей главе.

Нельзя недооценивать влияние проектных решений, которые мы уже приняли: размер наших входных данных, структуру кэширования и то, как мы разделяем наши обучающие и проверочные наборы, поскольку из всего этого складывается успех проекта или его провал. Вдобавок желательно пересматривать принятые решения позже, особенно работая над собственными проектами.

10.7. УПРАЖНЕНИЯ

1. Реализуйте программу, перебирающую экземпляр `LunaDataset` и измеряющую время, которое для этого потребуется. В целях экономии времени может иметь смысл перебирать лишь первые $N=1000$ образцов.
 - А. Сколько времени нужно для первого запуска?
 - Б. Сколько времени нужно для второго запуска?
 - В. Как очистка кэша влияет на время выполнения?
 - Г. Какой результат дает перебор *последних* $N=1000$ образцов в первый и второй раз?
2. Измените реализацию `LunaDataset`, чтобы рандомизировать список образцов в `__init__`. Очистите кэш и запустите модифицированную версию. Как это повлияет на время выполнения первого и второго запусков?

3. Отмените рандомизацию и закомментируйте декоратор `@functools.lru_cache(1, typed=True)` метода `getCt`. Очистите кэш и запустите модифицированную версию. Каким стало время выполнения?

10.8. РЕЗЮМЕ

- Часто код, необходимый для парсинга и загрузки необработанных данных, оказывается нетривиальным. Для этого проекта мы реализуем класс `Ct`, который загружает данные с диска и обеспечивает доступ к областям вокруг точек интереса.
- Кэширование бывает полезно, если парсинг и загрузка долго выполняются. Имейте в виду, что часть кэширования может выполняться в оперативной памяти, а часть лучше выполнять на диске. У каждого способа свое место в конвейере загрузки данных.
- Подклассы `PyTorch Dataset` используются для преобразования данных из их исходной формы в тензоры, подходящие для передачи в модель. С помощью этой функциональности мы можем интегрировать наши реальные данные с API-интерфейсами `PyTorch`.
- Подклассы `Dataset` должны реализовывать методы `__len__` и `__getitem__`. Можно использовать и другие вспомогательные методы, но это не обязательно.
- Правильное разделение данных на обучающий и проверочный наборы требует, чтобы мы убедились, что некий элемент данных не должен присутствовать сразу в двух наборах. Это достигается за счет согласованного порядка сортировки и отбора каждого десятого элемента в проверочный набор.
- Важно выполнять визуализацию данных. Возможность визуально исследовать данные позволяет много узнать об ошибках или проблемах. Для визуализации данных мы используем `Jupyter Notebook` и `Matplotlib`.

Обучение модели классификации обнаружению потенциальных опухолей

В этой главе

- ✓ Использование класса PyTorch DataLoader для загрузки данных.
- ✓ Реализация модели, выполняющей классификацию данных КТ.
- ✓ Заложение основы нашего приложения.
- ✓ Логирование и отображение метрик.

В предыдущих главах мы заложили основу для нашего проекта по обнаружению рака. Мы рассмотрели проблему с медицинской точки зрения, изучили основные источники данных, используемые в проекте, и преобразовали наши необработанные КТ-сканы в экземпляры `Dataset` PyTorch. Теперь, имея набор данных, мы можем легко использовать наши обучающие данные. Так и поступим!

11.1. БАЗОВАЯ МОДЕЛЬ И ЦИКЛ ОБУЧЕНИЯ

В этой главе мы сделаем две основные вещи. Мы начнем с создания модели классификации узелков и цикла обучения, тем самым заложив основу, которую далее на протяжении части II будем использовать для создания более крупного проекта. Загружать экземпляры `DataLoader` мы будем с помощью классов `Ct` и `LunaDataset`, которые реализовали в главе 10. Эти экземпляры, в свою очередь,

будут снабжать нашу модель классификации данными через циклы обучения и проверки.

В конце главы мы возьмем результаты выполнения обучающего цикла и обрисуем одну из самых сложных задач в этой части книги: как получить высококачественные результаты из беспорядочных ограниченных данных. В следующих главах мы рассмотрим, в чем ограничены наши данные, а также смягчим эти ограничения.

Вспомним нашу общую дорожную карту из главы 9 (мы продублировали ее ниже на рис. 11.1). Сейчас мы начнем работать над созданием модели, которая будет осуществлять этап 4 — классификацию. Напоминаем, что мы будем классифицировать кандидатов на узелки (в главе 14 мы создадим еще один классификатор, который будет отличать злокачественные узлы от доброкачественных). Это означает, что каждому образцу, который мы передадим модели, нужно будет присвоить метку. В данном случае метка будет *odule* или *non-odule*, поскольку каждый элемент данных соответствует одному кандидату.

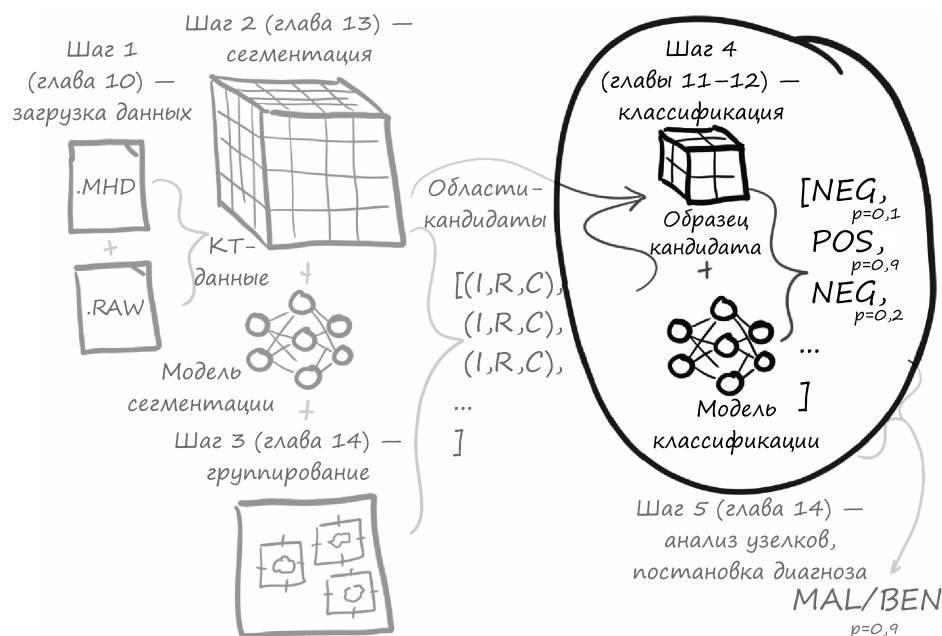


Рис. 11.1. Наш сквозной проект по выявлению рака легких. Выделена основная тема данной главы: этап 4 — классификация

Быстрый переход к сквозной версии значимой части вашего проекта — это важная веха. Наличие механизма, который бы работал достаточно хорошо для

аналитической оценки результатов, позволяет вам двигаться вперед в уверенности, что каждое изменение улучшает работу программы в целом, а плохие изменения можно будет отложить в сторонку! Не удивляйтесь, если при работе над вашими собственными проектами вам придется много экспериментировать. Достижение лучших результатов обычно требует значительной доработки и настройки.

Но прежде чем мы сможем перейти к экспериментированию, нужно заложить основу. На рис. 11.2 показано, как выглядит наш цикл обучения, используемый в части II. Он должен показаться вам в целом знакомым, поскольку аналогичный перечень этапов мы рассматривали в главе 5. Здесь для оценки нашего прогресса в обучении мы также будем применять проверочный набор, как обсуждалось в подразделе 5.5.3.

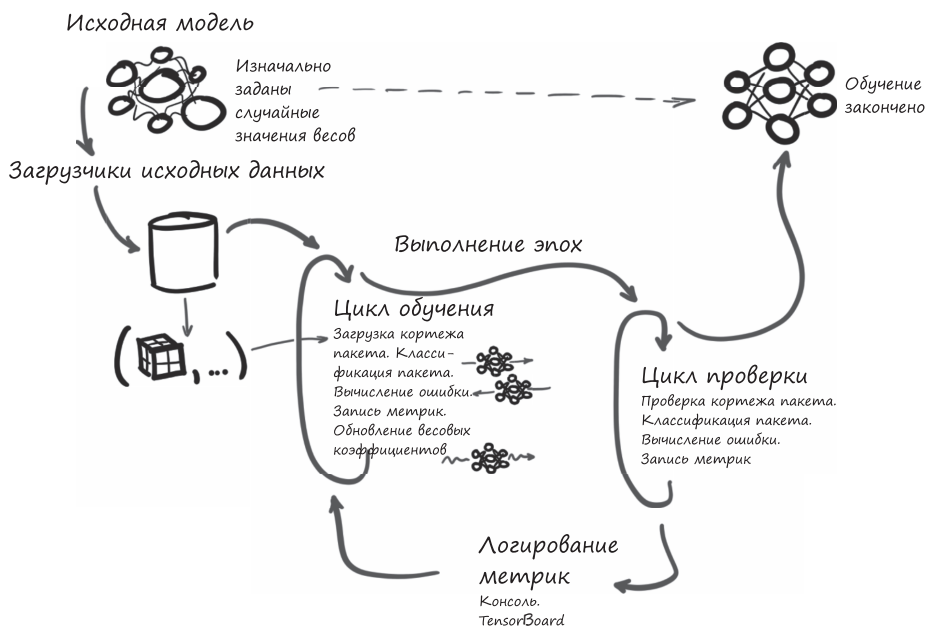


Рис. 11.2. Сценарий обучения и проверки, который мы реализуем в этой главе

Базовая структура того, что мы собираемся реализовать, выглядит следующим образом.

- Инициализация модели и загрузка данных.
- Запуск цикла по произвольно выбранному количеству эпох.
 - Перебор каждого пакета обучающих данных, возвращенного `LunaDataset`.
 - Загрузка соответствующего пакета данных в фоновом режиме загрузки данных.

- Передача пакета в модель классификации с целью получить результаты.
- Расчет потерь: разницы между полученным прогнозом и достоверными данными.
- Запись показателей производительности модели во временную структуру данных.
- Обновление весовых коэффициентов модели методом обратного распространения ошибки.
- Перебор всех пакетов проверочных данных (по аналогии с обучающим циклом).
- Загрузка соответствующего пакета проверочных данных (опять же в фоновом процессе).
- Классификация пакета и вычисление потерь.
- Запись информации о качестве работы модели, определенном при проверке данных.
- Вывод информации о прогрессе и производительности для этой эпохи.

Во время изучения кода этой главы обратите внимание на два основных различия между кодом, который мы создаем здесь, и кодом, который мы использовали для цикла обучения в части I. Для начала добавим в нашу программу больше структуры, поскольку проект в целом получился немного сложнее, чем то, что мы делали в предыдущих главах. Без дополнительной структуры код может быстро усложниться. В этом проекте наше основное обучающее приложение будет вызывать несколько грамотно организованных функций, и, кроме того, мы дополнительно выделим код, выполняющий отдельную задачу, в автономные модули Python.

В ваших проектах уровень структуры и дизайна должен соответствовать уровню сложности проекта. Если проект недостаточно структурирован, то вам будет трудно чисто проводить эксперименты, устранять неполадки или даже описывать то, что вы вообще делаете! С другой стороны, слишком *сложная* структура заставляет вас писать много ненужного кода и, скорее всего, замедлять работу, поскольку вам понадобится некоторое время на то, чтобы привыкнуть к этой структуре. Кроме того, у вас может возникнуть соблазн заняться инфраструктурой исключительно ради того, чтобы отложить подальше копание в сложном коде и достижение фактического прогресса в вашем проекте. Не попадайтесь в эту ловушку!

Еще одно большое различие между кодом этой главы и частью I заключается в сборе различных метрик о том, как продвигается обучение. Невозможно точно определить, как те или иные изменения влияют на обучение, если не логировать метрики. В следующей главе (не спойлер) мы также увидим, насколько важно собирать не просто метрики, а *правильные* метрики. В данной главе мы создадим инфраструктуру для получения этих метрик и с ее помощью будем вычислять

и отображать потери и процент правильно классифицированных образцов как в целом, так и по классам. Для начала этого достаточно, но более реалистичный набор показателей мы рассмотрим в главе 12.

11.2. ТОЧКА ВХОДА ПРИЛОЖЕНИЯ

Одно из существенных структурных отличий данной задачи от обучения, которое мы выполняли раньше, заключается в том, что в части II мы оборачиваем нашу работу в полноценное приложение командной строки. В нем будет парсинг аргументов командной строки, полнофункциональная команда `--help`, и ее можно будет легко запустить в самых разных средах. Все это позволит нам легко запускать процедуры обучения как из Jupyter, так и из оболочки Bash¹.

Функционал нашего приложения будет реализован через класс, чтобы мы могли создать экземпляр приложения и передать его куда-нибудь, если нужно. Это может упростить тестирование, отладку или вызов других программ Python. Мы можем вызвать приложение, обойдясь без запуска второго процесса на уровне ОС (в нашей книге мы не будем явным образом выполнять модульное тестирование, но созданная нами структура может быть полезна в реальных проектах, где применяется такое тестирование).

Возможность запустить обучение с помощью вызова функции или процесса на уровне ОС позволяет нам, например, обернуть вызовы функций в Jupyter Notebook, чтобы код можно было легко вызывать либо из интерфейса командной строки, либо из браузера (листинг 11.1).

Листинг 11.1. `code/p2_run_everything.ipynb`

```
# In[2]:w
def run(app, *argv):
    argv = list(argv)
    argv.insert(0, '--num-workers=4')
    log.info("Running: {}".format(app, argv))

    app_cls = importstr(*app.rsplit('.', 1))
    app_cls(argv).main()

    log.info("Finished: {}".format(app, argv))

# In[6]:
run('p2ch11.training.LunaTrainingApp', '--epochs=1')
```

Мы предполагаем, что у вас четырехъядерный восьмипоточный ЦП. Измените значение 4, если нужно

Это немного более чистый вызов директивы `__import__`

¹ На самом деле из любой оболочки, и если вы используете оболочку, отличную от Bash, то уже знаете об этом.

ПРИМЕЧАНИЕ

Обучение здесь предполагает, что вы работаете на рабочей станции с четырехъядерным процессором с восемью потоками, 16 Гбайт ОЗУ и графическим процессором с 8 Гбайт ОЗУ. Уменьшите значение `--batch-size`, если у вашего графического процессора меньше оперативной памяти, и `--num-workers`, если у вас меньше ядер процессора или оперативной памяти процессора.

Возьмем полустандартный шаблонный код. Для начала в конце файла добавим идиому `if main`, которая создает экземпляр объекта приложения и вызывает метод `main` (листинг 11.2).

Листинг 11.2. `training.py:386`

```
if __name__ == '__main__':
    LunaTrainingApp().main()
```

Теперь вы можете вернуться к началу файла и посмотреть на класс приложения и две функции, которые мы только что вызвали, `__init__` и `main`. Нам нужна возможность принимать аргументы командной строки, поэтому воспользуемся стандартной библиотекой `argparse` (<https://docs.python.org/3/library/argparse.html>) в функции `__init__` приложения. Обратите внимание, что мы можем передать инициализатору пользовательские аргументы, если захотим. Метод `main` будет основной точкой входа для базовой логики приложения (листинг 11.3).

Листинг 11.3. `training.py:31`, класс `LunaTrainingApp`

```
class LunaTrainingApp:
    def __init__(self, sys_argv=None):
        if sys_argv is None:
            sys_argv = sys.argv[1:]
        parser = argparse.ArgumentParser()
        parser.add_argument('--num-workers',
            help='Number of worker processes for background data loading',
            default=8,
            type=int,
        )
        # ... строка 63
        self.cli_args = parser.parse_args(sys_argv)
        self.time_str = datetime.datetime.now().strftime('%Y-%m-%d_%H.%M.%S')
        # ... строка 137
    def main(self):
        log.info("Starting {}, {}".format(type(self).__name__, self.cli_args))
```

Если вызывающая сторона не предоставляет аргументов, то мы получаем их из командной строки

С помощью временных меток мы будем идентифицировать обучающие запуски

Эта структура используется часто и пригодится в будущих проектах. В частности, парсинг аргументов в `__init__` позволяет нам настраивать приложение отдельно от его вызова.

В коде этой главы на сайте книги или в GitHub вы можете найти несколько дополнительных строк, в которых упоминается TensorBoard. Пока их можно проигнорировать; мы обсудим их подробно позже, в разделе 11.9.

11.3. ПРЕДВАРИТЕЛЬНАЯ НАСТРОЙКА И ИНИЦИАЛИЗАЦИЯ

Прежде чем мы сможем начать перебирать пакеты в пределах эпохи, необходимо выполнить некую работу по инициализации. Дело в том, что мы не можем обучить модель, если мы еще даже не создали ее экземпляр! Как видно из рис. 11.3, нам нужно сделать две основные вещи.

Первая, о которой мы только что говорили, — это инициализация нашей модели и оптимизатора; а вторая — инициализация экземпляров Dataset и DataLoader. Класс LunaDataset определит рандомизированный набор элементов данных, которые составят нашу эпоху обучения, а экземпляр DataLoader загрузит данные из общего набора и передаст их приложению.

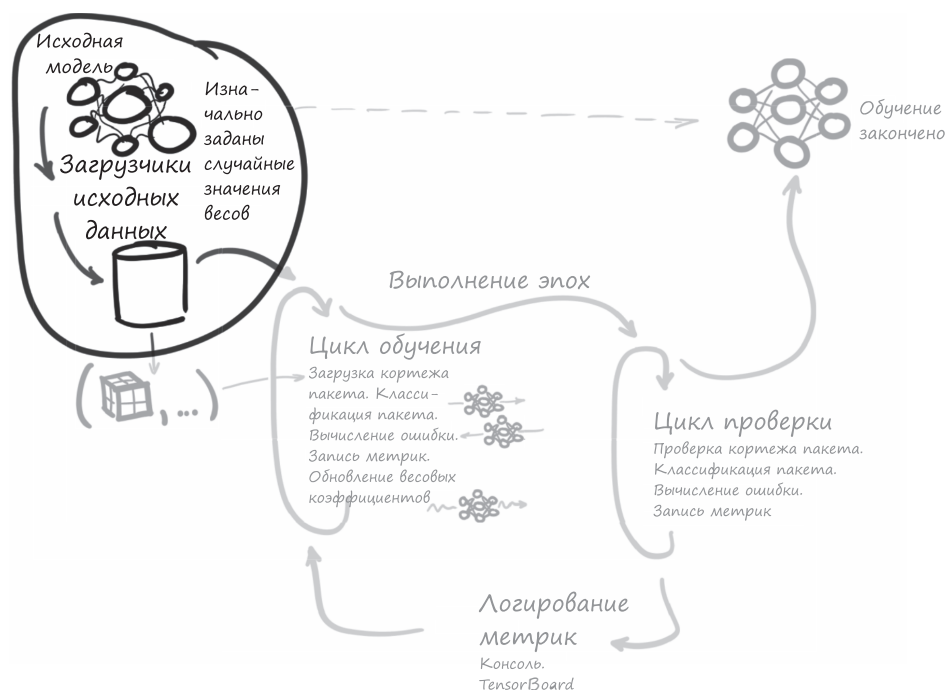


Рис. 11.3. Сценарий обучения и проверки, который мы реализуем в этой главе. Выделена область инициализации переменной перед циклом

11.3.1. Инициализация модели и оптимизатора

В этом подразделе мы рассматриваем детали `LunaModel` как «черный ящик». В разделе 11.4 мы подробно расскажем об их внутреннем устройстве. Вы можете изучить изменения в реализации и понять, как приблизить модель к желаемой цели, но, вероятно, лучше было бы сделать это после прочтения хотя бы главы 12.

Посмотрим, как выглядит наша отправная точка (листинг 11.4).

Листинг 11.4. `training.py:31`, класс `LunaTrainingApp`

```
class LunaTrainingApp:
    def __init__(self, sys_argv=None):
        # ... строка 70
        self.use_cuda = torch.cuda.is_available()
        self.device = torch.device("cuda" if self.use_cuda else "cpu")

        self.model = self.initModel()
        self.optimizer = self.initOptimizer()

    def initModel(self):
        model = LunaModel()
        if self.use_cuda:
            log.info("Using CUDA; {} devices.".format(torch.cuda.device_count()))
            if torch.cuda.device_count() > 1:  ← Обертка для модели
                model = nn.DataParallel(model)
                model = model.to(self.device)  ← Отправка параметров модели в ГП
            return model

    def initOptimizer(self):
        return SGD(self.model.parameters(), lr=0.001, momentum=0.99)
```

Обнаружение нескольких ГП

Если в системе, используемой для обучения, установлено более одного графического процессора, то мы задействуем класс `nn.DataParallel` для распределения работы между всеми графическими процессорами в системе, затем собираем и повторно синхронизируем обновления параметров и т. д. Этот метод почти полностью прозрачен с точки зрения как реализации модели, так и кода, использующего эту модель.

Предполагая, что значение `self.use_cuda` истинно, вызов `self.model.to(device)` перемещает параметры модели в ГП, настраивая свертки и другие вычисления с целью использовать ГП для тяжелой вычислительной работы. Важно сделать это перед созданием оптимизатора, поскольку в противном случае оптимизатору придется работать с объектами в ЦП, а не с объектами, скопированными в ГП.

В качестве оптимизатора мы будем использовать базовый стохастический градиентный спуск (SGD, <https://pytorch.org/docs/stable/optim.html#torch.optim.SGD>) с импульсом. Мы уже встречали этот оптимизатор в главе 5. Вспомним из части I, что в PyTorch имеется множество различных оптимизаторов. Мы не будем подробно

рассматривать большинство из них, но в официальной документации (<https://pytorch.org/docs/stable/optim.html#algorithms>) приведено немало ссылок на нужные документы.

DATAPARALLEL ПРОТИВ DISTRIBUTEDDATAPARALLEL

В этой книге мы обрабатываем случай использования нескольких графических процессоров с помощью класса `DataParallel`. Мы выбрали именно его, поскольку им легко обернуть уже имеющиеся модели. Но в целом этот способ применения нескольких графических процессоров не является самым эффективным и ограничен работой с оборудованием, имеющимся на одной машине.

В PyTorch также есть класс `DistributedDataParallel`, который рекомендуется использовать в случаях, когда вам нужно распределить работу между несколькими графическими процессорами или машинами. Правильно выполнить его настройку довольно непросто, и мы подозреваем, что подавляющее большинство наших читателей не увидят никакой пользы в сложности, поэтому в данной книге мы не будем рассматривать `DistributedDataParallel`. Если вы хотите узнать больше, то мы предлагаем прочитать официальную документацию: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html.

SGD довольно часто используется в качестве первого оптимизатора. В некоторых задачах SGD может работать плохо, но такие задачи относительно редки. Аналогично скорость обучения 0,001 и импульс 0,9 — достаточно безопасные стартовые параметры. Опыт свидетельствует, что SGD с этими значениями хорошо показал себя в довольно широком круге проектов, и вы также можете легко попробовать задать скорость обучения 0,01 или 0,0001, если что-то не работает сразу.

Мы не говорим, что какое-либо из этих значений лучше других для нашего случая, и нам пока рано заниматься поиском идеала. Систематическое изменение значений скорости обучения, импульса, размера сети и других подобных параметров конфигурации называется *поиском по гиперпараметрам*. Есть и другие, более насущные вопросы, которые нам необходимо рассмотреть в следующих главах. Разобравшись с ними, мы сможем приступить к тонкой настройке этих значений. Как мы упоминали в пункте «Тестирование других оптимизаторов» в главе 5, существуют и другие, более экзотические оптимизаторы, но их принципы работы и объяснение связанных с ними компромиссов достаточно сложны для этой книги, кроме, может быть, `torch.optim.Adam`.

11.3.2. Передача данных загрузчикам

Созданный нами в предыдущей главе класс `LunaDataset` играет роль моста между любыми имеющимися у нас необработанными данными и несколько

более структурированным миром тензоров, которые нужны для работы PyTorch. Например, `torch.nn.Conv3d` (<https://pytorch.org/docs/stable/nn.html#conv3d>) ожидает пятимерный ввод (N, C, D, H, W): количество элементов данных, число каналов на элемент, глубину, высоту и ширину. Эта структура далека от исходного 3D, которое у нас было после КТ!

Вспомним вызов `ct_t.unsqueeze(0)` в `LunaDataset.__getitem__` из предыдущей главы. В нем у данных есть четвертое измерение — канал. Из главы 4 мы помним, что у изображения RGB всего три канала, по одному на каждый цвет. В астрономических данных могут быть десятки каналов для различных слоев электромагнитного спектра: гамма-лучей, рентгеновских лучей, ультрафиолетового света, видимого света, инфракрасного излучения, микроволн и/или радиоволн. Поскольку компьютерная томография работает на одной длине волны, число каналов равно 1.

Вспомним также из части I, что обучение на единичных элементах данных обычно неэффективно с точки зрения вычислительных ресурсов, поскольку большинство платформ могут выполнять больше параллельных вычислений, чем требуется модели для обработки одного обучающего или проверочного элемента. Решение состоит в том, чтобы сгруппировать элементы данных в кортеж, как показано на рис. 11.4; благодаря этому можно обрабатывать несколько элементов одновременно. Пятое измерение (N) позволяет различать элементы данных в пределах пакета.

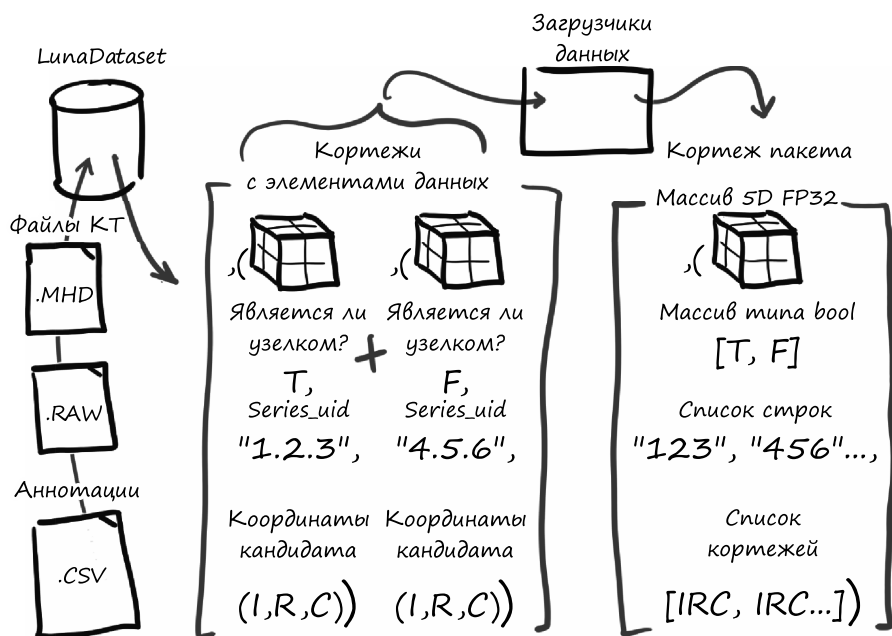


Рис. 11.4. Примеры кортежей сопоставляются в один пакетный кортеж внутри загрузчика данных

Удобно, что нам не нужно реализовывать пакетную обработку, поскольку класс PyTorch `DataLoader` выполняет эту задачу за нас. Мы уже построили преобразование из КТ-сканов в тензоры PyTorch с помощью класса `LunaDataset`, поэтому осталось лишь подключить наш набор данных к загрузчику данных (листинг 11.5).

Листинг 11.5. `training.py:89, LunaTrainingApp.initTrainDl`

```
def initTrainDl(self):
    train_ds = LunaDataset(      ← Пользовательский набор данных
        val_stride=10,
        isValSet_bool=False,
    )

    batch_size = self.cli_args.batch_size
    if self.use_cuda:
        batch_size *= torch.cuda.device_count()

    train_dl = DataLoader(      ← Готовый класс
        train_ds,
        batch_size=batch_size,  ← Разбиение на пакеты выполняется автоматически
        num_workers=self.cli_args.num_workers,
        pin_memory=self.use_cuda, ← Область памяти перемещается в ГП
    )

    return train_dl

# ... строка 137
def main(self):
    train_dl = self.initTrainDl()
    val_dl = self.initValDl()    ← Загрузчик проверочных данных работает аналогично обучающему
```

В дополнение к пакетной обработке отдельных образцов загрузчики данных также могут обеспечивать параллельную загрузку данных с помощью отдельных процессов и общей памяти. Все, что нам нужно сделать, — это указать `num_workers=...` при создании экземпляра загрузчика данных, а остальное сделается «за кулисами». Каждый рабочий процесс производит готовые пакеты, как показано на рис. 11.4. Это помогает убедиться, что в графические процессоры поступает достаточно данных. Наши экземпляры `validation_ds` и `validation_dl` выглядят одинаково, за исключением очевидного `isValSet_bool=True`.

Когда мы выполняем перебор в цикле `for batch_tup in self.train_dl:`, нам не нужно ждать, пока загрузится каждый `Ct`, из него будут сгруппированы данные и т. д. Вместо этого мы немедленно получим уже загруженный `batch_tup`, а рабочий процесс будет освобожден в фоновом режиме, чтобы начать загрузку другого пакета для использования в более поздней итерации. Использование функций загрузки данных PyTorch позволяет ускорить большинство проектов, поскольку мы можем переложить загрузку и обработку данных на ГП.

11.4. ПЕРВЫЙ СКВОЗНОЙ ДИЗАЙН НЕЙРОННОЙ СЕТИ

Просторы проектирования сверточной нейронной сети, способной обнаруживать опухоли, фактически безграничны. К счастью, в последнее десятилетие или около того ученые потратили немало времени на изучение эффективных моделей распознавания изображений. В основном эти модели ориентированы на 2D-изображения, но общие архитектурные идеи хорошо переносятся в 3D, поэтому существует множество проверенных проектов, которые мы можем использовать в качестве отправной точки. Это полезно, поскольку наша первая архитектура вряд ли будет лучшей в мире, но в данный момент мы будем стремиться к тому, чтобы она «работала достаточно хорошо».

Дизайн сети будет основан на том, что мы использовали в главе 8. Нам придется несколько обновить модель, поскольку входные данные у нас трехмерные и мы добавим ряд усложняющих деталей, но общая структура, показанная на рис. 11.5, должна показаться вам знакомой. Аналогично работа, которую мы сделаем в этом проекте, станет хорошей основой для ваших будущих проектов, и чем

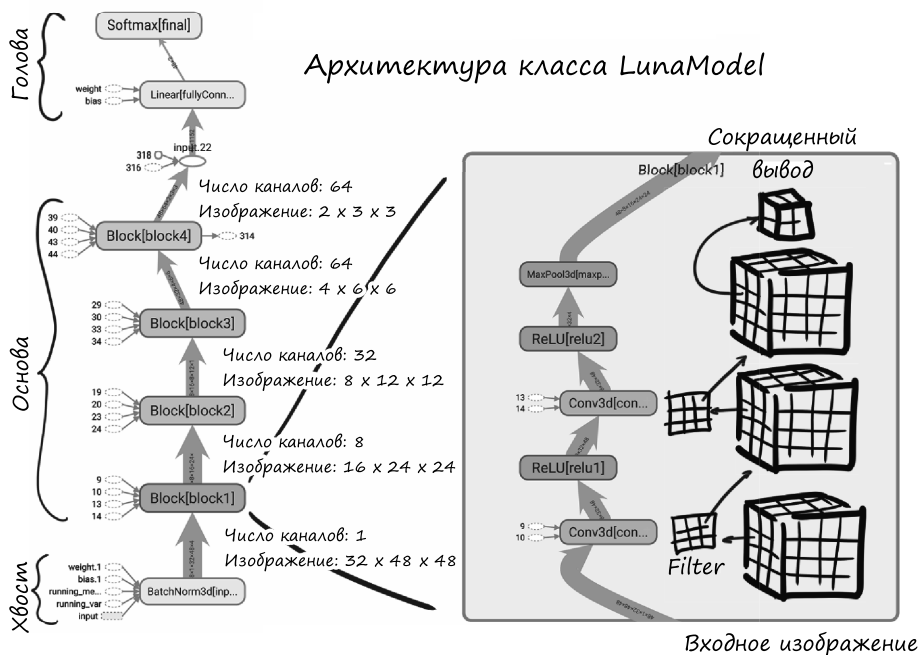


Рис. 11.5. Архитектура класса `LunaModel`, состоящая из хвоста пакетной нормализации, четырех блоков основы и головы, включающей линейный слой и `Softmax`

больше проектов по классификации или сегментации вы выполните, тем больше вам придется адаптировать эту базу, чтобы она подходила для конкретной задачи. Разберем данную архитектуру, начиная с четырех повторяющихся блоков, составляющих основную часть сети.

11.4.1. Основы свертки

В моделях классификации часто используется структура, состоящая из хвоста, основы (или тела) и головы. *Хвост* — это первые несколько слоев, которые обрабатывают входные данные сети. Структура или организация первых слоев часто отличается от слоев в остальной части сети, поскольку они должны преобразовывать входные данные в форму, ожидаемую основной частью. Здесь мы используем простой слой пакетной нормализации, хотя часто в хвосте также бывают сверточные слои. Они часто служат для агрессивного уменьшения размера изображения, но в нашем случае размер изображения уже мал, так что нам это не требуется.

В *теле* сети, как правило, содержится большая часть слоев, которые обычно располагаются в виде нескольких *блоков*. В каждом блоке одинаковый (или по крайней мере схожий) набор слоев, хотя часто размер входных данных и количество фильтров у блоков различаются. Мы будем использовать блок, состоящий из двух сверток 3×3 , за каждой из которых следует функция активации и операция максимального объединения в конце блока. Проиллюстрируем это на расширенном представлении рис. 11.5, помеченном как Block[block].

В листинге 11.6 приведена реализация блока в коде.

Листинг 11.6. model.py:67, класс LunaBlock

```
class LunaBlock(nn.Module):
    def __init__(self, in_channels, conv_channels):
        super().__init__()
        self.conv1 = nn.Conv3d(
            in_channels, conv_channels, kernel_size=3, padding=1, bias=True,
        )
        self.relu1 = nn.ReLU(inplace=True) 1((C05-1))
        self.conv2 = nn.Conv3d(
            conv_channels, conv_channels, kernel_size=3, padding=1, bias=True,
        )
        self.relu2 = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool3d(2, 2)

    def forward(self, input_batch):
        block_out = self.conv1(input_batch)
        block_out = self.relu1(block_out)
        block_out = self.conv2(block_out)
        block_out = self.relu2(block_out)

        return self.maxpool(block_out)
```

Как вариант, здесь можно
обращаться к функциям API

Теперь голова сети берет выходные данные из основной части и преобразует их в желаемую выходную форму. В сверточных сетях часто выполняется выравнивание промежуточных выходных данных и их передача на полносвязный слой. Для ряда сетей имеет смысл включить еще и второй полносвязный слой, но так чаще делают в задачах классификации, в которых отображаемые объекты имеют крупную структуру (например, сравнение легковых автомобилей с грузовиками, имеющими колеса, фары, решетку, двери и т. д.), и для проектов с большим количеством классов. Поскольку мы занимаемся лишь бинарной классификацией и нам не нужна дополнительная сложность, мы используем только один слой выравнивания.

Подобная структура может стать хорошим первым строительным блоком для сверточной сети. Существуют и более сложные структуры, но во многих проектах их применение излишне с точки зрения как сложности реализации, так и вычислительных требований. Лучше начать с простого и усложнять только тогда, когда в этом есть очевидная необходимость.

Выполняемая внутри блока свертка показана в 2D на рис. 11.6. Поскольку берется лишь небольшая часть большого изображения, мы игнорируем отступы. (Обратите внимание, что функция активации ReLU не показана, поскольку ее применение не меняет размеры изображения.)

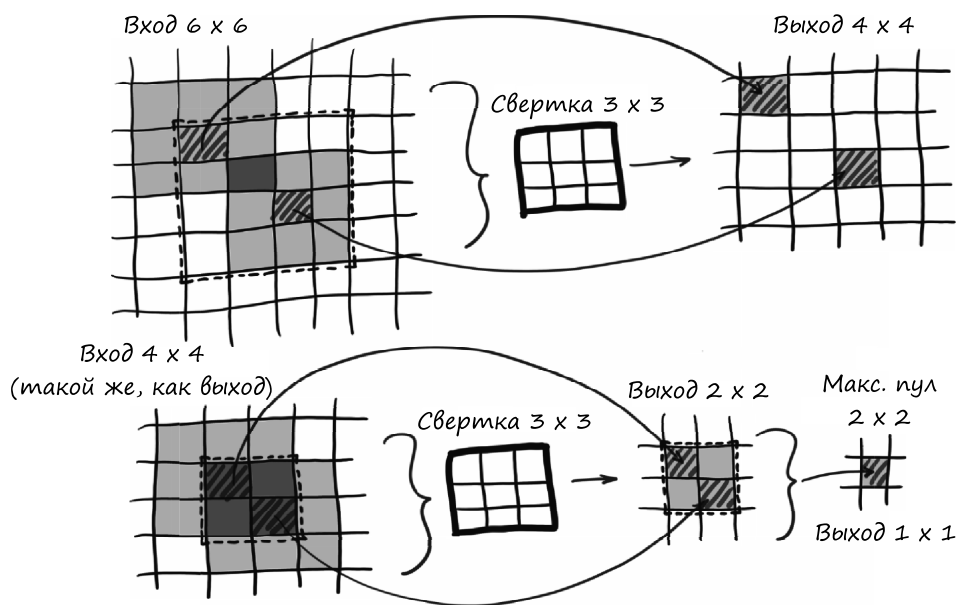


Рис. 11.6. Сверточная архитектура блока LunaModel, состоящая из двух сверток 3×3 , за которыми следует по максимальный пул. Последний пиксель имеет рецептивное поле размером 6×6

Проанализируем информационный поток между нашими входными вокселями и выходным вокселем. Нам нужно четкое представление о том, как наши выходные данные будут реагировать на изменение входных. Было бы неплохо просмотреть главу 8, особенно разделы с 8.1 по 8.3, просто чтобы убедиться, что вы на 100 % уверены в понимании того, как работает свертка.

В блоке мы используем свертки $3 \times 3 \times 3$. Одиночная свертка $3 \times 3 \times 3$ имеет рецептивное поле размером $3 \times 3 \times 3$, что почти тавтологично. Вводятся 27 вокселей, а выходит один.

Интереснее становится, когда мы используем две свертки $3 \times 3 \times 3$, расположенные друг за другом. Наложение сверточных слоев позволяет входным данным влиять на окончательный выходной воксель (или пиксель) сильнее, чем предполагает размер ядра свертки.

Если выходной воксель подается в другое ядро $3 \times 3 \times 3$ в качестве одного из краевых вокселей, то часть входных данных первого слоя будет находиться за пределами области $3 \times 3 \times 3$ второго ввода. Конечный результат двух сложенных слоев имеет *эффективное рецептивное поле* $5 \times 5 \times 5$. Это означает, что взятые вместе слои действуют так же, как один сверточный слой большего размера.

Иными словами, каждый сверточный слой $3 \times 3 \times 3$ добавляет дополнительный одновоксельный слой по краю рецептивного поля. Мы увидим это, если проследим за стрелкой на рис. 11.6 в обратном направлении: выход 2×2 имеет рецептивное поле 4×4 , которое, в свою очередь, имеет рецептивное поле 6×6 . При этом два сложенных слоя $3 \times 3 \times 3$ используют меньше параметров, чем полная свертка $5 \times 5 \times 5$ (и, следовательно, вычисление также выполняется быстрее).

Результат двух сложенных сверток подается в максимальный пул $2 \times 2 \times 2$, а это означает, что мы берем эффективное поле $6 \times 6 \times 6$, отбрасываем семь восьмых данных и получаем поле $5 \times 5 \times 5$, которое произвело наибольшее значение¹. Теперь у «отброшенных» входных вокселей все еще есть шанс внести свой вклад, поскольку максимальный пул, который приходится на один выходной воксель, имеет перекрывающееся входное поле, поэтому они тоже могут повлиять на конечный результат.

Обратите внимание, что рецептивное поле сжимается с каждым сверточным слоем, но мы используем *дополненные* свертки, которые добавляют вокруг изображения виртуальную границу в один пиксель. При этом размеры входного и выходного изображения не меняются.

Слои nn.ReLU аналогичны тем, которые мы рассматривали в главе 6. Выходы больше 0.0 останутся без изменений, а выходы меньше 0.0 будут приведены к нулю.

Этот блок будет повторяться несколько раз, формируя основу нашей модели.

¹ Помните, что на самом деле мы работаем в 3D, несмотря на 2D-иллюстрацию.

11.4.2. Полная модель

Посмотрим на полную реализацию модели (листинг 11.7). Мы пропустим определение блока, так как мы только что видели его в листинге 11.6.

Листинг 11.7. model.py:13, класс LunaModel

```
class LunaModel(nn.Module):
    def __init__(self, in_channels=1, conv_channels=8):
        super().__init__()

        self.tail_batchnorm = nn.BatchNorm3d(1)  ← Хвост

        self.block1 = LunaBlock(in_channels, conv_channels)
        self.block2 = LunaBlock(conv_channels, conv_channels * 2)
        self.block3 = LunaBlock(conv_channels * 2, conv_channels * 4)
        self.block4 = LunaBlock(conv_channels * 4, conv_channels * 8)

        self.head_linear = nn.Linear(1152, 2)
        self.head_softmax = nn.Softmax(dim=1)
```

Основа

Голова

Хвост здесь относительно прост. Мы собираемся нормализовать наши входные данные с помощью класса `nn.BatchNorm3d`, который, как мы видели в главе 8, сдвигает и масштабирует наши входные данные так, что их среднее значение становится равно 0, а стандартное отклонение — 1. Таким образом, несколько странная единица измерения Хаунсфилда (HU), в которых измерены входные данные, остальной части сети не будет видна. Это несколько произвольный выбор. Мы знаем, какие единицы используются на входе, и знаем ожидаемые значения для соответствующих тканей, поэтому можно было бы довольно легко реализовать фиксированную схему нормализации. Неясно, какой подход будет лучше¹.

Наша основа состоит из четырех повторяющихся блоков, а реализация блока вынесена в отдельный подкласс `nn.Module`, который мы видели ранее в листинге 11.6. Поскольку каждый блок заканчивается операцией максимального пула $2 \times 2 \times 2$, после четырех слоев мы уменьшим разрешение изображения в 16 раз в каждом измерении. Вспомним из главы 10, что наши данные возвращаются партиями размером $32 \times 48 \times 48$, которые к концу этой части сети придут к размеру $2 \times 3 \times 3$.

Наконец, наш хвост — это просто полносвязный слой, за которым следует вызов `nn.Softmax`. Функция `Softmax` полезна для задач классификации с одной меткой, и у нее есть несколько приятных свойств: она ограничивает выходные данные значениями между 0 и 1, к тому же относительно нечувствительна к абсолютному

¹ Поэтому в следующей главе есть упражнение для экспериментирования и с тем и с другим!

диапазону входных данных (имеют значение только *относительные* значения входных данных), и это позволяет нашей модели выражать степень уверенности в ответе.

Сама функция относительно проста. Каждое значение из ввода используется для экспонирования e , а полученный ряд значений затем делится на сумму всех результатов возведения в степень. Вот как это выглядит в простой и неоптимизированной форме на чистом Python:

```
>>> logits = [1, -2, 3]
>>> exp = [e ** x for x in logits]
>>> exp
[2.718, 0.135, 20.086]

>>> softmax = [x / sum(exp) for x in exp]
>>> softmax
[0.118, 0.006, 0.876]
```

Конечно, для нашей модели мы используем версию `nn.Softmax` для PyTorch, поскольку она умеет работать с пакетами и тензорами и будет выполнять автоградиацию быстро и так, как ожидалось.

Усложнение: преобразование данных из свертки в вектор

Продолжим определение модели, введя усложнение. Нельзя просто так взять и передать выходные данные `self.block4` в полносвязный слой, поскольку этот выход представляет собой изображение $2 \times 3 \times 3$ для каждой точки данных с 64 каналами, а полносвязные слои ожидают в качестве входных данных одномерный вектор (ну, технически они ожидают *пакет* одномерных векторов, который представляет собой двумерный массив, однако несоответствие остается в любом случае). Посмотрим на метод `forward` (листинг 11.8).

Листинг 11.8. `model.py:50, LunaModel.forward`

```
def forward(self, input_batch):
    bn_output = self.tail_batchnorm(input_batch)

    block_out = self.block1(bn_output)
    block_out = self.block2(block_out)
    block_out = self.block3(block_out)
    block_out = self.block4(block_out)

    conv_flat = block_out.view(
        block_out.size(0),  ← Размер пакета
        -1,
    )
    linear_output = self.head_linear(conv_flat)

    return linear_output, self.head_softmax(linear_output)
```

Обратите внимание: перед передачей данных в полносвязный слой мы должны сгладить их с помощью функции `view`. Поскольку эта операция не имеет состояния (у нее нет параметров, управляющих ее поведением), мы можем просто выполнить операцию в функции `forward`. Это чем-то похоже на функциональные интерфейсы, которые мы обсуждали в главе 8. Почти в каждой модели, использующей свертки и производящей классификации, регрессии или другие выходные данные, не связанные с изображениями, в голове сети применяется нечто аналогичное.

Метод `forward` возвращает как необработанные *логиты*, так и вероятности, созданные `softmax`. Впервые мы говорили о логитах в подразделе 7.2.6: это числовые значения, созданные сетью до того, как они были нормализованы в вероятности уровнем `softmax`. Звучит сложно, однако на самом деле логиты — это просто исходные данные для слоя `softmax`. Они могут иметь любые действительные входные данные, и `softmax` сжимает их до диапазона 0–1.

Мы будем использовать логиты при расчете `nn.CrossEntropyLoss` во время обучения¹, а вероятности — когда нужно классифицировать элементы данных. Такая небольшая разница между тем, что используется для обучения, и тем, что применяется в производственной среде, довольно распространена, особенно когда разница между двумя выходными данными представляет собой простую функцию без сохранения состояния, такую как `softmax`.

Инициализация

Наконец, поговорим об инициализации параметров сети. Чтобы добиться хорошей производительности модели, веса, смещения и другие параметры сети должны обладать определенными свойствами. Представим себе вырожденный случай, когда все веса сети больше 1 (и у нас нет остаточных связей). В таком случае повторное умножение на эти веса приведет к тому, что выходные данные слоя будут расти по мере прохождения данных через слои сети. Аналогично, вес меньше 1 приведет к уменьшению и исчезновению выходных данных всех слоев. Такие же соображения применимы к градиентам в обратном проходе.

Обеспечить правильное поведение выходных данных слоя можно с помощью множества методов нормализации. Один из самых простых — убедиться, что веса сети инициализированы таким образом, чтобы промежуточные значения и градиенты не становились ни неоправданно малыми, ни неоправданно большими. Как мы обсуждали в главе 8, в данном вопросе PyTorch нам не помогает, поэтому нам нужно выполнить инициализацию самостоятельно. Мы можем рассматривать следующую функцию `_init_weights` как шаблонную, поскольку точные детали не особенно важны (листинг 11.9).

¹ В пользу этого подхода есть аргументы, связанные с числовой стабильностью. Точное распространение градиентов через экспоненту, вычисленную с помощью 32-битных чисел с плавающей запятой, может быть проблематичным.

Листинг 11.9. model.py:30, LunaModel_init_weights

```
def _init_weights(self):
    for m in self.modules():
        if type(m) in {
            nn.Linear,
            nn.Conv3d,
        }:
            nn.init.kaiming_normal_(
                m.weight.data, a=0, mode='fan_out', nonlinearity='relu',
            )
            if m.bias is not None:
                fan_in, fan_out = \
                    nn.init._calculate_fan_in_and_fan_out(m.weight.data)
                bound = 1 / math.sqrt(fan_out)
                nn.init.normal_(m.bias, -bound, bound)
```

11.5. ОБУЧЕНИЕ И ПРОВЕРКА МОДЕЛИ

Теперь пришло время взять части, с которыми мы работали, и собрать их в нечто реализуемое. Этот тренировочный цикл должен быть вам знаком — циклы, подобные показанному на рис. 11.7, вы видели в главе 5.

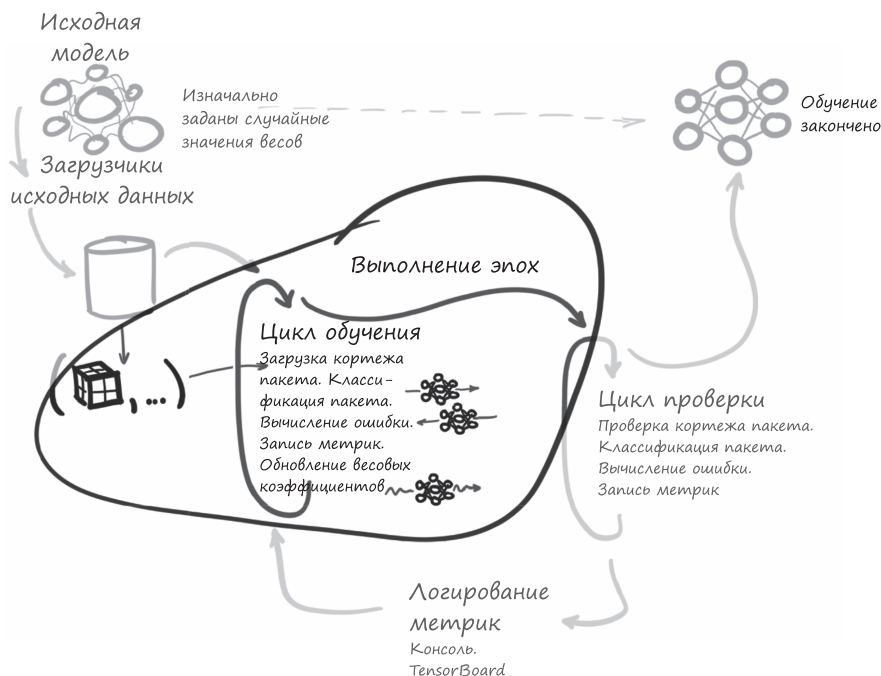


Рис. 11.7. Сценарий обучения и проверки, который мы реализуем в этой главе, уделяя особое внимание вложенным циклам для каждой эпохи и пакетам в эпохе

Код здесь относительно компактен (функция `doTraining` состоит всего из 12 операторов, но здесь она длиннее из-за ограничений длины строки) (листинг 11.10).

Листинг 11.10. `training.py:137, LunaTrainingApp.main`

```
def main(self):
    # ... строка 143
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        trnMetrics_t = self.doTraining(epoch_ndx, train_dl)
        self.logMetrics(epoch_ndx, 'trn', trnMetrics_t)

    # ... строка 165
    def doTraining(self, epoch_ndx, train_dl):
        self.model.train()
        trnMetrics_g = torch.zeros(
            METRICS_SIZE,
            len(train_dl.dataset),
            device=self.device,
        )

        batch_iter = enumerateWithEstimate(
            train_dl,
            "E{} Training".format(epoch_ndx),
            start_ndx=train_dl.num_workers,
        )
        for batch_ndx, batch_tup in batch_iter:
            self.optimizer.zero_grad()
            loss_var = self.computeBatchLoss(
                batch_ndx,
                batch_tup,
                train_dl.batch_size,
                trnMetrics_g
            )

            loss_var.backward()
            self.optimizer.step()

        self.totalTrainingSamples_count += len(train_dl.dataset)

    return trnMetrics_g.to('cpu')
```

← Инициализация пустого массива метрик

← Проход по пакету с измерением времени

← Высвобождение остаточных тензоров

← Этот метод мы подробнее рассмотрим в следующем разделе

← Обновление весовых коэффициентов

Ниже описаны основные отличия от тренировочных циклов, которые мы рассматривали в предыдущих главах.

- Тензор `trnMetrics_g` собирает подробные метрики для каждого класса во время обучения. Для более крупных проектов, таких как наш, это может быть очень удобным.
- Мы не перебираем напрямую загрузчик данных `train_dl`. Мы используем `enumerateWithEstimate`, чтобы указать предполагаемое время завершения. Это не имеет решающего значения — просто стилистический выбор.

- Фактическое вычисление потерь помещается в метод `calculateBatchLoss`. Опять же это не строго обязательно, но повторное использование кода обычно является плюсом.

Почему мы обернули `enumerate` дополнительным функционалом, обсудим в подразделе 11.7.2, а на данный момент предположим, что это то же самое, что `enumerate(train_dl)`.

Тензор `trnMetrics_g` должен передавать информацию о том, как модель ведет себя для каждой точки данных от функции `computeBatchLoss` до функции `logMetrics`. Посмотрим на `computeBatchLoss`. А `logMetrics` мы рассмотрим после того, как закончим с остальной частью основного обучающего цикла.

11.5.1. Функция `calculateBatchLoss`

Функция `calculateBatchLoss` вызывается и обучающим, и проверочным циклами. Как следует из названия, она вычисляет потери в партии данных. Кроме того, функция вычисляет и записывает информацию о выходных данных модели для каждой точки данных. Это позволяет нам вычислять такие вещи, как процент правильных ответов на класс, что дает возможность оттачивать области, в которых модель испытывает трудности.

Конечно, основная задача этой функции — передача пакета в модель и вычисление потерь для каждого пакета. Мы используем класс `CrossEntropyLoss` (<https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss>) точно так же, как в главе 7. Распаковка кортежа пакета, перемещение тензоров в графический процессор и вызов модели вам уже знакомы (листинг 11.11).

Листинг 11.11. `training.py:225, .computeBatchLoss`

```
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g):
    input_t, label_t, _series_list, _center_list = batch_tup

    input_g = input_t.to(self.device, non_blocking=True)
    label_g = label_t.to(self.device, non_blocking=True)

    logits_g, probability_g = self.model(input_g)

    loss_func = nn.CrossEntropyLoss(reduction='none')
    loss_g = loss_func(
        logits_g,
        label_g[:,1],
    )
    # ... строка 238
    return loss_g.mean()
```

reduction='none' возвращает
потери для точки данных

Индекс класса

Рекомбинация потерь точки
данных в одно значение

Здесь мы *не* используем поведение по умолчанию для получения значения потерь, усредненного по пакету. Вместо этого мы получаем тензор значений потерь, по одному на точку данных. Это позволяет нам отслеживать отдельные потери; то есть мы можем агрегировать их по своему усмотрению (например, по классам). Как это работает, мы увидим через мгновение. На данный момент мы вернем среднее значение этих потерь на точку данных, что эквивалентно потерям для пакета. Если вы не хотите вести статистику по выборке, то можно использовать усредненные потери по пакету. Так ли это, в значительной степени зависит от вашего проекта и целей.

Как только это будет сделано, мы выполним свои обязательства перед вызывающей функцией с точки зрения того, что требуется для выполнения обратного распространения и обновления веса. Однако прежде, чем сделать это, мы также хотим записать нашу статистику по каждой выборке для потомков (и последующего анализа). Для этого мы будем использовать переданный параметр `metrics_g` (листинг 11.12).

Листинг 11.12. `training.py:26`

```
METRICS_LABEL_NDX=0
METRICS_PRED_NDX=1
METRICS_LOSS_NDX=2
METRICS_SIZE = 3

# ... строка 225
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g):
    # ... строка 238
    start_ndx = batch_ndx * batch_size
    end_ndx = start_ndx + label_t.size(0)

    metrics_g[METRICS_LABEL_NDX, start_ndx:end_ndx] = \
        label_g[:,1].detach()
    metrics_g[METRICS_PRED_NDX, start_ndx:end_ndx] = \
        probability_g[:,1].detach()
    metrics_g[METRICS_LOSS_NDX, start_ndx:end_ndx] = \
        loss_g.detach()

    return loss_g.mean()
```

Именованные индексы массива
объявляются на уровне модуля

Мы используем отсоединение,
так как ни одна из наших
метрик не должна удерживать
градиенты

Опять же это потери по всему пакету

Записывая метку, прогноз и потери для каждой обучающей (а позже и проверочной) точки данных, мы получаем массу подробной информации, которую можем использовать для исследования поведения нашей модели. Сейчас мы сосредоточимся на составлении статистики по классам, но с помощью этой информации могли бы легко найти выборку, классифицированную наиболее неправильно, и начать исследовать почему. Опять же для некоторых проектов подобная информация будет менее интересной, но хорошо помнить, что у вас есть такие варианты.

11.5.2. Цикл проверки работает аналогично

Цикл проверки на рис. 11.8 очень похож на обучающий, но несколько упрощен. Ключевое отличие состоит в том, что проверка выполняет только чтение. В частности, возвращаемое значение потерь не используется, а веса не обновляются.

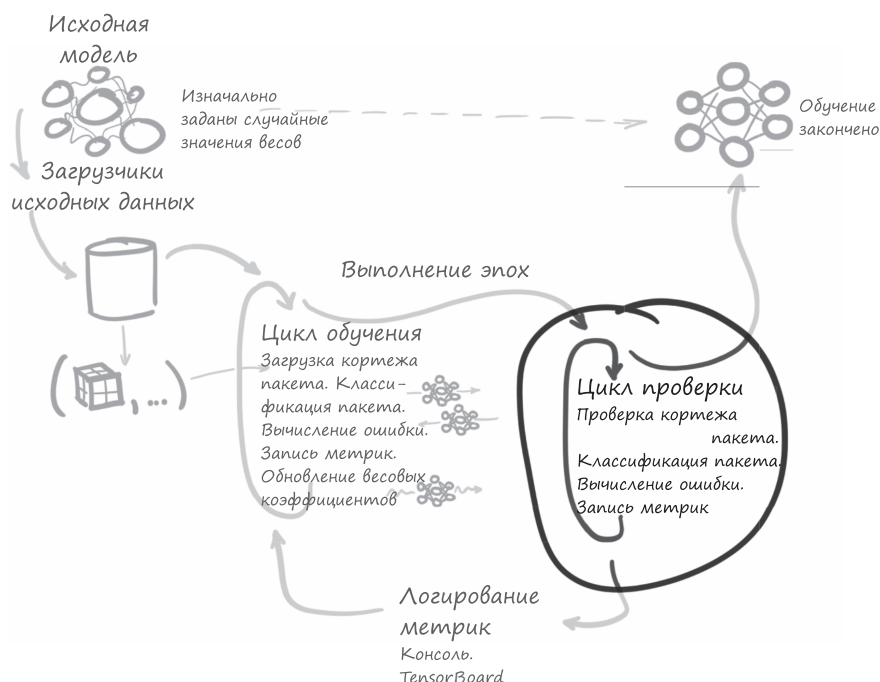


Рис. 11.8. Сценарий обучения и проверки, который мы реализуем в этой главе. Выделен цикл проверки

Между началом и концом вызова функции в модели не должно ничего измениться. Кроме того, этот код работает немного быстрее благодаря использованию контекстного менеджера `with torch.no_grad()`, явно информирующего PyTorch о том, что не нужно вычислять градиенты (листинг 11.13).

Листинг 11.13. training.py:137, LunaTrainingApp.main

```
def main(self):
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        # ... строка 157
        valMetrics_t = self.doValidation(epoch_ndx, val_dl)
        self.logMetrics(epoch_ndx, 'val', valMetrics_t)

# ... строка 203
```



```
def doValidation(self, epoch_ndx, val_dl):
    with torch.no_grad():
        self.model.eval()  ← Отключение логики обучения
        valMetrics_g = torch.zeros(
            METRICS_SIZE,
            len(val_dl.dataset),
            device=self.device,
        )

        batch_iter = enumerateWithEstimate(
            val_dl,
            "E{} Validation ".format(epoch_ndx),
            start_ndx=val_dl.num_workers,
        )
        for batch_ndx, batch_tup in batch_iter:
            self.computeBatchLoss(
                batch_ndx, batch_tup, val_dl.batch_size, valMetrics_g)

    return valMetrics_g.to('cpu')
```

Когда не требуется обновлять значения весовых коэффициентов сети (так как это идет вразрез с самой идеей проверочного набора!), нам не нужно использовать потери, возвращенные из метода `calculateBatchLoss`, равно как и применять оптимизатор. Внутри цикла остался лишь вызов `calculateBatchLoss`. Обратите внимание, что эта функция в качестве побочного эффекта по-прежнему собирает метрики в переменную `valMetrics_g`, но мы никак не используем общие потери на пакет, возвращаемые `calculateBatchLoss`.

11.6. ВЫВОД МЕТРИК ПРОИЗВОДИТЕЛЬНОСТИ

Последнее, что нужно сделать в каждой эпохе, — записать в лог показатели производительности для этой эпохи. На рис. 11.9 видно, что после логирования метрик мы возвращаемся к циклу обучения и выполняем следующую эпоху. Запись результатов и прогресса по мере обучения весьма важна, поскольку, если обучение идет не по плану (или «не сходится», на языке глубокого обучения), нам нужно об этом знать и перестать тратить время на обучение модели, которая не работает. Но даже если катастрофы не произошло, хорошо иметь возможность следить за тем, как ведет себя ваша модель.

Ранее для логирования прогресса в каждой эпохе мы собирали результаты в тензорах `trnMetrics_g` и `valMetrics_g`. Каждый из них теперь содержит все необходимое для вычисления правильных процентов и средних потерь по классам для обучения и проверки. Эти действия в каждой эпохе выполняются часто, хотя и не всегда в этом есть смысл. В следующих главах мы увидим, как можно менять размер эпох, чтобы получать обратную связь о ходе обучения с разумной скоростью.

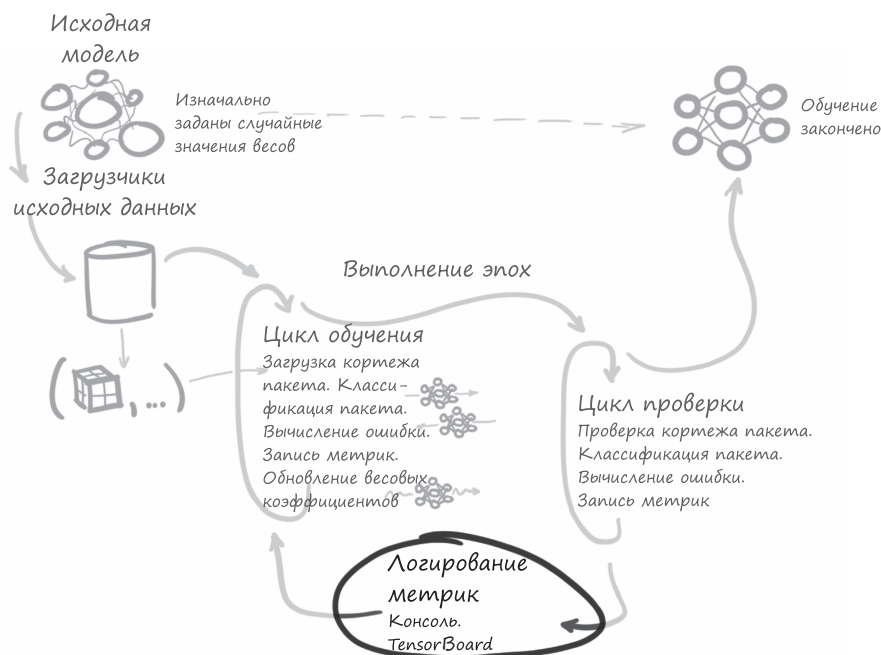


Рис. 11.9. Сценарий обучения и проверки, который мы реализуем в этой главе. Выделен этап логирования метрик в конце каждой эпохи

11.6.1. Функция logMetrics

Рассмотрим высокоуровневую структуру функции `logMetrics`. Ее сигнатура выглядит следующим образом (листинг 11.14).

Листинг 11.14. `training.py:251, LunaTrainingApp.logMetrics`

```
def logMetrics(
    self,
    epoch_ndx,
    mode_str,
    metrics_t,
    classificationThreshold=0.5,
):
```

Параметр `epoch_ndx` используется исключительно для отображения результатов во время их логирования. Аргумент `mode_str` определяет, для чего предназначены метрики: для обучения или проверки.

Функция использует тензор `trnMetrics_t` или `valMetrics_t`, передающийся в параметр `metrics_t`. Вспомните, что оба варианта — это тензоры значений с плавающей запятой, которые мы заполнили данными в функции `computeBatchLoss`,

а затем передали обратно в ЦП прямо перед тем, как вернуть их из `doTraining` и `doValidation`. У обоих тензоров три строки и столько же столбцов, сколько у нас точек данных (обучающих или проверочных, в зависимости от ситуации). Напомним, что эти три строки соответствуют следующим константам (листинг 11.15).

Листинг 11.15. training.py:26

```
METRICS_LABEL_NDX=0  ← Объявляются на уровне модуля
METRICS_PRED_NDX=1
METRICS_LOSS_NDX=2
METRICS_SIZE = 3
```

МАСКИРОВКА ТЕНЗОРОВ И БУЛЕВО ИНДЕКСИРОВАНИЕ

Маскировка тензоров — это распространенный паттерн, который может оказаться трудным для понимания, если вы не сталкивались с ним раньше. Возможно, вы знакомы с библиотекой NumPy и используемыми в ней *маскированными массивами*. Так вот, тензорные маски работают так же.

Если вы не знакомы с маскированными массивами, то в документации NumPy (<http://mng.bz/XPra>) хорошо описано, как они работают. В PyTorch намеренно используются те же синтаксис и семантика, что и в NumPy.

Составление маски

Нужно создать маски, с помощью которых метрики будут ограничены только элементами данных с узлами или без узлов (то есть положительными или отрицательными). Вдобавок мы подсчитаем общее количество элементов на класс, а также тех элементов, которые мы классифицировали правильно (листинг 11.16).

Листинг 11.16. training.py:264, LunaTrainingApp.logMetrics

```
negLabel_mask = metrics_t[METRICS_LABEL_NDX] <= classificationThreshold
negPred_mask = metrics_t[METRICS_PRED_NDX] <= classificationThreshold

posLabel_mask = ~negLabel_mask
posPred_mask = ~negPred_mask
```

Мы не используем оператор `assert`, поэтому знаем, что все значения, хранящиеся в `metrics_t[METRICS_LABEL_NDX]`, принадлежат множеству `{0.0, 1.0}`, то есть просто `True` или `False`. По сравнению со значением `classificationThreshold`, которое по умолчанию равняется `0.5`, мы получаем массив бинарных значений, где значение `True` соответствует метке «не узелок» для данного элемента данных.

Мы делаем аналогичное сравнение, чтобы создать `negPred_mask`, но важно помнить следующее: значения `METRICS_PRED_NDX` — это положительные прогнозы,

созданные нашей моделью, и они могут принимать дробные значения от 0,0 до 1,0 включительно. Это не меняет оператор сравнения, но означает, что фактическое значение может быть близко к 0,5. Положительные маски — это просто обратные отрицательные маски.

ПРИМЕЧАНИЕ

В других проектах тоже можно применять аналогичные подходы, но важно понимать: сокращения данных допустимы именно потому, что мы решаем задачу бинарной классификации. Если в вашем следующем проекте будет более двух классов или элементы данных будут принадлежать к нескольким классам одновременно, вам придется использовать более сложную логику для создания подобных масок.

Затем мы используем эти маски для вычисления некоторых статистических данных для каждой метки и сохраняем их в словаре `metrics_dict` (листинг 11.17).

Листинг 11.17. training.py:270, LunaTrainingApp.logMetrics

```
neg_count = int(negLabel_mask.sum())
pos_count = int(posLabel_mask.sum())

neg_correct = int((negLabel_mask & negPred_mask).sum())
pos_correct = int((posLabel_mask & posPred_mask).sum())

metrics_dict = {}
metrics_dict['loss/all'] = \
    metrics_t[METRICS_LOSS_NDX].mean()
metrics_dict['loss/neg'] = \
    metrics_t[METRICS_LOSS_NDX, negLabel_mask].mean()
metrics_dict['loss/pos'] = \
    metrics_t[METRICS_LOSS_NDX, posLabel_mask].mean()

metrics_dict['correct/all'] = (pos_correct + neg_correct) \
    / np.float32(metrics_t.shape[1]) * 100
metrics_dict['correct/neg'] = neg_correct / np.float32(neg_count) * 100
metrics_dict['correct/pos'] = pos_correct / np.float32(pos_count) * 100
```

Преобразование в целое
число Python

Обход целочисленного
деления путем
преобразования
в np.float32

Сперва мы вычисляем средние потери за всю эпоху. Поскольку потеря — это единственная метрика, которая во время обучения должна минимизироваться, мы всегда хотим иметь возможность отслеживать ее. Затем мы ограничиваем усреднение потерь только элементами данных с отрицательной меткой, используя только что созданную маску `negLabel_mask`. То же самое делаем с положительными потерями. Такое вычисление потерь по классам бывает полезно, если один класс постоянно труднее классифицировать, чем другой, и полученные результаты могут помочь в проведении исследований и улучшений.

В конце расчета мы вычислили процент элементов данных, которые мы классифицировали правильно, а также процент правильных результатов по каждой метке. Для отображения в процентах потребуется умножить значение на 100.

Как и в случае с потерями, эти числа могут помочь нам понять, что и где можно улучшить. По окончании вычислений мы логируем результаты с помощью трех вызовов `log.info` (листинг 11.18).

Листинг 11.18. `training.py:289, LunaTrainingApp.logMetrics`

```
log.info(
    ("E{} {}:8} {loss/all:.4f} loss, "
     + "{correct/all:-5.1f}% correct, "
    ).format(
        epoch_ndx,
        mode_str,
        **metrics_dict,
    )
)
log.info(
    ("E{} {}:8} {loss/neg:.4f} loss, "
     + "{correct/neg:-5.1f}% correct ({neg_correct:} of {neg_count:})"
    ).format(
        epoch_ndx,
        mode_str + '_neg',
        neg_correct=neg_correct,
        neg_count=neg_count,
        **metrics_dict,
    )
)
log.info(  ← Положительное логирование выполняется так же, как и отрицательное
    # ... строка 319
)
```

В первом логе хранятся значения, рассчитанные по всем элементам данных с меткой `/all`, а отрицательные (не узелковые) и положительные (узелковые) значения помечены `/neg` и `/pos` соответственно. В коде не приведен третий оператор регистрации для положительных значений; он идентичен второму, за исключением замены `neg` на `pos` во всех случаях.

11.7. ЗАПУСК ОБУЧЕНИЯ

Когда основа сценария `training.py` готова, его можно запускать. Он инициализирует и обучает нашу модель, выводя в процессе информацию о том, насколько хорошо идет обучение. Процесс обучения работает в фоновом режиме, пока мы подробно рассматриваем реализацию модели. Надеюсь, когда мы закончим, у вас появятся осязаемые результаты.

Сценарий запускается из корневого каталога с кодом. В нем должны быть подкаталоги с именами `p2ch11`, `util` и т. д. В используемой среде Python должны быть установлены все библиотеки, перечисленные в файле `requirements.txt`. Как только эти библиотеки будут готовы, мы сможем запустить следующую команду:

```
$ python -m p2ch11.training
Starting LunaTrainingApp,
  Namespace(batch_size=256, channels=8, epochs=20, layers=3, num_workers=8)
<p2ch11.dsets.LunaDataset object at 0x7fa53a128710>: 495958 training samples
<p2ch11.dsets.LunaDataset object at 0x7fa537325198>: 55107 validation samples
Epoch 1 of 20, 1938/216 batches of size 256
E1 Training ----/1938, starting
E1 Training 16/1938, done at 2018-02-28 20:52:54, 0:02:57
...
```

← Это командная строка для Linux/Bash. Пользователям Windows, вероятно, нужно вызывать Python по-другому, в зависимости от используемого метода установки

Мы также предоставляем файл Jupyter Notebook, который содержит вызовы обучающего приложения (листинг 11.19).

Листинг 11.19. code/p2_run_everything.ipynb

```
# In[5]:
run('p2ch11.prepcache.LunaPrepCacheApp')

# In[6]:
run('p2ch11.training.LunaTrainingApp', '--epochs=1')
```

Если вам кажется, что эпоха 1 выполняется слишком долго (более 10 или 20 минут), то это может быть связано с необходимостью подготовки кэшированных данных, которые нужны классу `LunaDataset`. Подробнее о кэшировании мы писали в подразделе 10.5.1. В числе упражнений для главы 10 было написание сценария для эффективного предварительного заполнения кэша. Вы также можете взять готовый файл `prepcache.py`, который именно это и делает, и вызвать его с помощью команды `python -m p2ch11.prepcache`. Поскольку мы обновляем файлы `dsets.py` в каждой главе, кэширование нужно также повторять для каждой главы. Это несколько неэффективно с точки зрения использования ресурсов, однако позволяет сохранять порядок в коде каждой главы. В будущих проектах мы рекомендуем задействовать кэш активнее и чаще.

После начала обучения нужно убедиться, что имеющиеся вычислительные ресурсы используются так, как ожидалось. Простой способ определить наличие узкого места в загрузке данных или вычислениях — подождать несколько секунд после того, как сценарий начнет обучение (сообщение вида `E1 Training 16/7750, done at...`), а затем проверить `top` и `nvidia-smi`:

- если восемь рабочих процессов Python загружают более 80 % ЦП, то, вероятно, вам необходимо подготовить кэш (мы это знаем заранее, поскольку предварительно сделали реализацию такой, чтобы у проекта не было узких мест ЦП, но в жизни так не бывает);
- если `nvidia-smi` сообщает, что значение `GPU-Util` более 80 %, то ГП загружен работой. Мы обсудим некоторые стратегии эффективного ожидания в подразделе 11.7.2.

Наша цель — насытить ГП работой, то есть мы хотим использовать как можно больше его вычислительной мощности для быстрого завершения эпох. Одна карта NVIDIA GTX 1080 Ti должна выполнять эпоху менее чем за 15 минут. Так как наша модель относительно проста, процессор не настолько загружен, чтобы стать узким местом. При работе с моделями с большей глубиной (или бóльшим объемом вычислений в целом) обработка каждого пакета будет занимать больше времени, что увеличит объем работы ЦП, который мы можем выполнить до того, как ГП начнет простаивать в ожидании новых данных.

11.7.1. Необходимые для обучения данные

Если количество точек данных меньше 495 958 в обучающем наборе или 55 107 в проверочном, то можно выполнить проверку работоспособности, чтобы убедиться, что для вычислений использовались все данные. В ваших будущих проектах стоит проверять, возвращает ли ваш набор данных ожидаемое количество элементов данных.

Для начала взглянем на базовую структуру нашего каталога `data-unversioned/part2/luna/`:

```
$ ls -lp data-unversioned/part2/luna/
subset0/
subset1/
...
subset9/
```

Затем удостоверимся, что у нас есть один файл `.mhd` и один файл `.raw` для каждого UID:

```
$ ls -lp data-unversioned/part2/luna/subset0/
1.3.6.1.4.1.14519.5.2.1.6279.6001.105756658031515062000744821260.mhd
1.3.6.1.4.1.14519.5.2.1.6279.6001.105756658031515062000744821260.raw
1.3.6.1.4.1.14519.5.2.1.6279.6001.108197895896446896160048741492.mhd
1.3.6.1.4.1.14519.5.2.1.6279.6001.108197895896446896160048741492.raw
...
```

и что общее количество файлов правильное:

```
$ ls -l data-unversioned/part2/luna/subset?/* | wc -l
1776
$ ls -l data-unversioned/part2/luna/subset0/* | wc -l
178
...
$ ls -l data-unversioned/part2/luna/subset9/* | wc -l
176
```

Если все сходится, но что-то по-прежнему не работает, то обратитесь с вопросом на сайте Manning LiveBook (<https://livebook.manning.com/book/deep-learning-with-pytorch/chapter-11>), и, надемся, кто-то сможет помочь вам разобраться.

11.7.2. Интерлюдия: функция `enumerateWithEstimate`

Работа с глубоким обучением всегда сопряжена с томительным ожиданием. В реальном мире вам придется сидеть без дела, уставившись в стену или в экран и пытаясь ускорить процесс силой мысли (это не получится, но вы можете пожарить яичницу на графическом процессоре), одним словом — *скука*.

Единственное, что может быть хуже, чем сидеть и смотреть на мигающий курсор, который не движется больше часа, — это полный экран вот таких сообщений:

```
2020-01-01 10:00:00,056 INFO training batch 1234
2020-01-01 10:00:00,067 INFO training batch 1235
2020-01-01 10:00:00,077 INFO training batch 1236
2020-01-01 10:00:00,087 INFO training batch 1237
...etc...
```

По крайней мере, тихо мигающий курсор не захламляет окно прокрутки!

Пока идет обучение, мы всегда задаем себе вопрос: «Успею ли я пойти и налить себе воды?», который постепенно превращается в «Успею ли я...»:

- «...сварить кофе?»;
- «...поужинать?»;
- «...поужинать в Париже?».

Ответить на эти насущные вопросы мы сможем с помощью функции `enumerateWithEstimate`. Она используется следующим образом:

```
>>> for i, _ in enumerateWithEstimate(list(range(234)), "sleeping"):
...     time.sleep(random.random())
...
11:12:41,892 WARNING sleeping ----/234, starting
11:12:44,542 WARNING sleeping   4/234, done at 2020-01-01 11:15:16, 0:02:35
11:12:46,599 WARNING sleeping   8/234, done at 2020-01-01 11:14:59, 0:02:17
11:12:49,534 WARNING sleeping  16/234, done at 2020-01-01 11:14:33, 0:01:51
11:12:58,219 WARNING sleeping  32/234, done at 2020-01-01 11:14:41, 0:01:59
11:13:15,216 WARNING sleeping  64/234, done at 2020-01-01 11:14:43, 0:02:01
11:13:44,233 WARNING sleeping 128/234, done at 2020-01-01 11:14:35, 0:01:53
11:14:40,083 WARNING sleeping ----/234, done at 2020-01-01 11:14:40
>>>
```

Эти восемь строк были выведены спустя 200 итераций продолжительностью около двух минут. Даже с учетом большой дисперсии `random.random()`

функция имела неплохую оценку после 16 итераций (менее десяти секунд). В циклах с более постоянной синхронизацией оценки стабилизируются еще быстрее.

С точки зрения поведения функция `enumerateWithEstimate` практически идентична стандартной функции `enumerate` (разница в том, что наша функция возвращает генератор, а `enumerate` возвращает объект `<enumerate object at 0x...>`) (листинг 11.20).

Листинг 11.20. `util.py:143,def enumerateWithEstimate`

```
def enumerateWithEstimate(
    iter,
    desc_str,
    start_ndx=0,
    print_ndx=4,
    backoff=None,
    iter_len=None,
):
    for (current_ndx, item) in enumerate(iter):
        yield (current_ndx, item)
```

Интересной эту функцию делают побочные эффекты (в частности, логирование). Чтобы не создавать в книге путаницу в попытках охватить каждую деталь реализации, мы разместили пояснения в строке документации функции (<https://github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/util/util.py#L143>), где вы можете почитать о параметрах функции и посмотреть на реализацию.

Проекты глубокого обучения могут занимать очень много времени. Примерное понимание сроков обучения позволяет вам разумно использовать свободное время, а заодно позволяет понять, когда что-то не работает должным образом (или вообще не работает), если ожидаемое время завершения намного больше, чем планировалось.

11.8. ОЦЕНКА МОДЕЛИ: 99,7 % ПРАВИЛЬНЫХ ОТВЕТОВ — ЭТО ОТЛИЧНЫЙ РЕЗУЛЬТАТ, НЕ ТАК ЛИ?

Рассмотрим выходные результаты сценария обучения. Напоминаем, что мы запустили его с помощью команды `python -m p2ch11.training`:

```
E1 Training ----/969, starting
...
E1 LunaTrainingApp
E1 trn      2.4576 loss,  99.7% correct
...
E1 val      0.0172 loss,  99.8% correct
...
```

По истечении эпохи 1 обучения как обучающий, так и проверочный набор показывают не менее 99,7 % правильных результатов. Это очень хорошо! Можете погладить себя по голове и довольно улыбнуться. Мы победил рак!.. Так ведь?

Хм... нет.

Рассмотрим вывод эпохи 1 более подробно и без сокращений:

```
E1 LunaTrainingApp
E1 trn      2.4576 loss,  99.7% correct,
E1 trn_neg  0.1936 loss,  99.9% correct (494289 of 494743)
E1 trn_pos  924.34 loss,   0.2% correct (3 of 1215)
...
E1 val      0.0172 loss,  99.8% correct,
E1 val_neg  0.0025 loss, 100.0% correct (494743 of 494743)
E1 val_pos  5.9768 loss,   0.0% correct (0 of 1215)
```

В проверочном наборе узелковые точки на 100 % правильные, а настоящие узелки на 100 % неверны. Сеть просто не видит ни одного узелка! Значение 99,7 % просто означает, что лишь около 0,3 % точек данных являются узелками.

После десяти эпох ситуация не намного лучше:

```
E10 LunaTrainingApp
E10 trn      0.0024 loss,  99.8% correct
E10 trn_neg  0.0000 loss, 100.0% correct
E10 trn_pos  0.9915 loss,   0.0% correct
E10 val      0.0025 loss,  99.7% correct
E10 val_neg  0.0000 loss, 100.0% correct
E10 val_pos  0.9929 loss,   0.0% correct
```

Результат классификации остается прежним — ни один из узелков (то есть положительных точек) не был идентифицирован правильно. Интересно, что потери `val_pos` снижаются, но соответствующего увеличения потерь `val_neg` нет. Это означает, что сеть чему-то учится. К сожалению, она учится очень, очень медленно.

Хуже того, именно этот режим отказа является самым опасным в реальном мире! Ни в коем случае нельзя классифицировать опухоль как безобидную структуру, поскольку пациент может остаться без правильного диагноза и возможного лечения, в котором он мог бы нуждаться. Во всех проектах важно понимать последствия неправильной классификации, так как потенциальный риск ошибки влияет на то, как вы разрабатываете, обучаете и оцениваете свою модель. Мы обсудим это более подробно в следующей главе.

Но прежде нужно несколько усовершенствовать наши инструменты, чтобы сделать результаты более понятными. Мы не сомневаемся, что огромные простыни чисел милы вашему сердцу, как и всем нам, но порой картинки стоят тысячи слов. Построим графики некоторых метрик.

11.9. ПОСТРОЕНИЕ ГРАФИКОВ ДЛЯ МЕТРИК ОБУЧЕНИЯ С ПОМОЩЬЮ TENSORBOARD

Воспользуемся инструментом под названием TensorBoard. Это быстрый и простой способ вывести метрики из цикла обучения и представить в виде красивых графиков. Он позволит нам следить за изменением метрик в динамике, а не только за отдельными значениями в отдельных эпохах. Визуальное представление всегда позволяет узнать, является ли то или иное значение исключением либо частью тренда.

Вы можете спросить: «Минуточку, а разве TensorBoard не часть проекта TensorFlow? Что он делает здесь, в книге по PyTorch?»

Ну да, этот инструмент — часть другой структуры глубокого обучения, но наш принцип гласит: «Используй то, что работает». Нет причин ограничивать себя и не применять некий инструмент только потому, что он связан с другим проектом. Разработчики и PyTorch, и TensorBoard согласны с этой мыслью и потому совместными усилиями добавили его официальную поддержку в PyTorch. Этот инструмент великолепен, он имеет несколько простых в использовании API PyTorch, которые позволяют подключать к нему данные практически из любого места для быстрого и удобного отображения. Работая в сфере глубокого обучения, вы, вероятно, будете *часто* видеть (и применять) TensorBoard.

Если вы выполняли примеры из глав, то у вас на диске уже должны быть готовые для отображения данные. Посмотрим, как запустить TensorBoard и что он может нам показать.

11.9.1. Запуск TensorBoard

По умолчанию наш обучающий скрипт записывает данные метрик в подкаталог `run/`. Просмотрев содержимое каталога из оболочки Bash, вы можете увидеть нечто наподобие этого:

<code>\$ ls -lA runs/p2ch11/</code>	
<code>total 24</code>	
<code>drwxrwxr-x 2 elis elis 4096 Sep 15 13:22 2020-01-01_12.55.27-trn-dlwpt/</code>	Одна эпоха, полученная после запуска
<code>drwxrwxr-x 2 elis elis 4096 Sep 15 13:22 2020-01-01_12.55.27-val-dlwpt/</code>	
<code>drwxrwxr-x 2 elis elis 4096 Sep 15 15:14 2020-01-01_13.31.23-trn-dwlpt/</code>	
<code>drwxrwxr-x 2 elis elis 4096 Sep 15 15:14 2020-01-01_13.31.23-val-dwlpt/</code>	Более поздний запуск на десяти эпохах

Для работы TensorBoard установите Python-пакет `tensorflow` (<https://pypi.org/project/tensorflow>). Поскольку мы не собираемся использовать TensorFlow на максимум, можно установить стандартный пакет только для ЦП. Если у вас уже есть другая версия TensorBoard, то можно использовать и ее. Убедитесь,

что работаете в правильном каталоге, или перейдите туда с помощью команды `../path/to/tensorboard --logdir runs/`. На самом деле не имеет значения, откуда вызывать команду, если вы используете аргумент `--logdir`, чтобы указать, где хранятся ваши данные. Рекомендуется разделить ваши данные по отдельным папкам, так как TensorBoard несколько загромождается, если вы проведете более 10 или 20 экспериментов. По мере продвижения вам придется решать, как лучше всего организовывать вывод данных для каждого проекта. Не бойтесь перемещать данные постфактум, если нужно.

Запустим TensorBoard:

```
$ tensorboard --logdir runs/
```

```
2020-01-01 12:13:16.163044: I tensorflow/core/platform/cpu_feature_guard.cc:140] ←
```

```
  Your CPU supports instructions that this TensorFlow binary was not
  ➡ compiled to use: AVX2 FMA 1((C017-2))
```

```
TensorBoard 1.14.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

У вас сообщения могут быть другими, но это нормально

После этого вы сможете открыть в браузере адрес <http://localhost:6006> и увидеть главную панель графиков¹. На рис. 11.10 показано, как она выглядит.

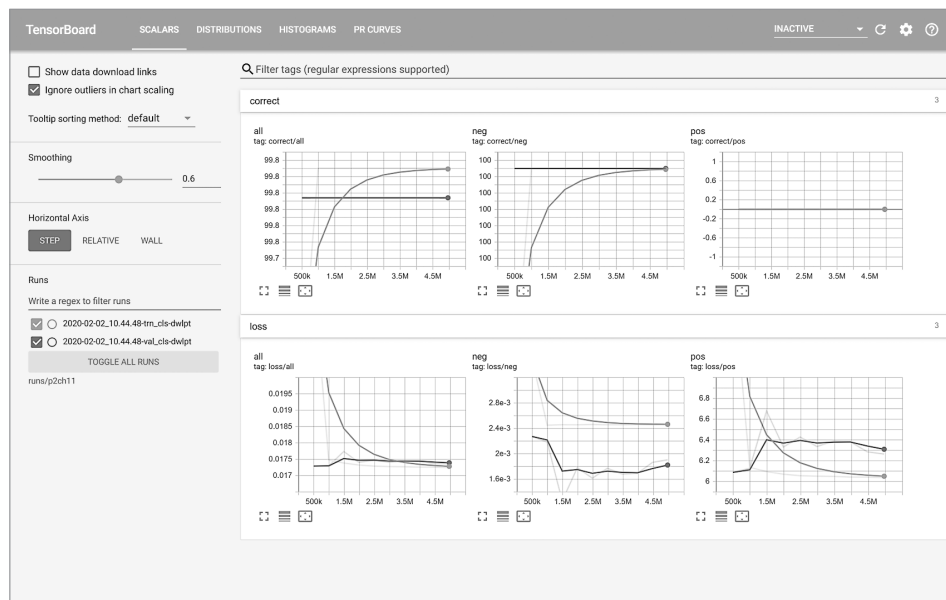


Рис. 11.10. Основной пользовательский интерфейс TensorBoard с параметрами обучающих и проверочных прогонов

¹ Если вы проводите обучение не на локальной машине, то вам необходимо заменить `localhost` соответствующим именем хоста или IP-адресом.

В верхней части окна браузера вы должны увидеть оранжевый заголовок. В его правой части находятся виджеты настроек, ссылка на репозиторий GitHub и т. п. Они нам пока не нужны. В левой части заголовка есть вкладки с разными типами данных, в вашем случае следующие:

- Scalars — скаляры (вкладка по умолчанию);
- Histograms — гистограммы;
- PR Curves — кривые точности-отзыва.

На рисунке также есть вкладка Distributions и вторая вкладка пользовательского интерфейса (справа от Scalars на рис. 11.10). Эти элементы нам не понадобятся. Сейчас перейдите на вкладку Scalars.

Слева приведены элементы управления параметрами отображения, а также список отображаемых запусков. Параметр сглаживания может быть полезен, если вы работаете с сильно зашумленными данными. Сглаживание позволяет уменьшить колебания и более ясно определить общую тенденцию. Исходные несглаженные данные по-прежнему будут видны на заднем плане в виде блеклой линии того же цвета. Она показана на рис. 11.11, хотя при черно-белой печати увидеть ее может быть сложно.

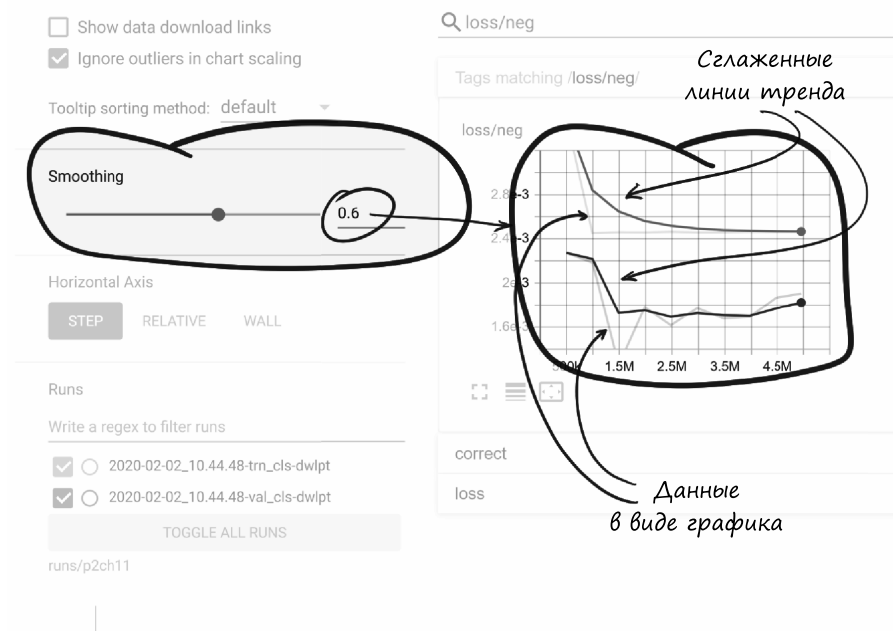


Рис. 11.11. Боковая панель TensorBoard со значением сглаживания 0,6 и отображением двух запусков

Если вы запускали сценарий обучения несколько раз, то можно выбрать, данные какого запуска отображать. При рендеринге слишком большого количества запусков графики могут стать неразборчивыми, поэтому стоит отменить отображение тех запусков, которые в данный момент вам неинтересны.

Если вы хотите удалить запуск навсегда, то данные можно удалить с диска во время работы TensorBoard. Вы можете сделать это, чтобы избавиться от неудачных экспериментов, ошибочных или просто устаревших данных.

Количество запусков может расти довольно быстро, поэтому часто бывает полезно удалять лишнее, а также переименовывать или перемещать нужные запуски, которые особенно интересны, в отдельный каталог, чтобы случайно не удалить их. Чтобы удалить и обучающий, и проверочный запуски, выполните следующее (соответствующим образом изменив номер главы, дату и время):

```
$ rm -rf runs/p2ch11/2020-01-01_12.02.15_*
```

Помните: удаление запусков приведет к тому, что данные, которые находятся ниже в списке, переместятся вверх и получат новые цвета.

Теперь рассмотрим то, ради чего мы скачали TensorBoard: красивые графики! Основная часть экрана должна быть заполнена данными сбора метрик обучения и проверки, как показано на рис. 11.12.

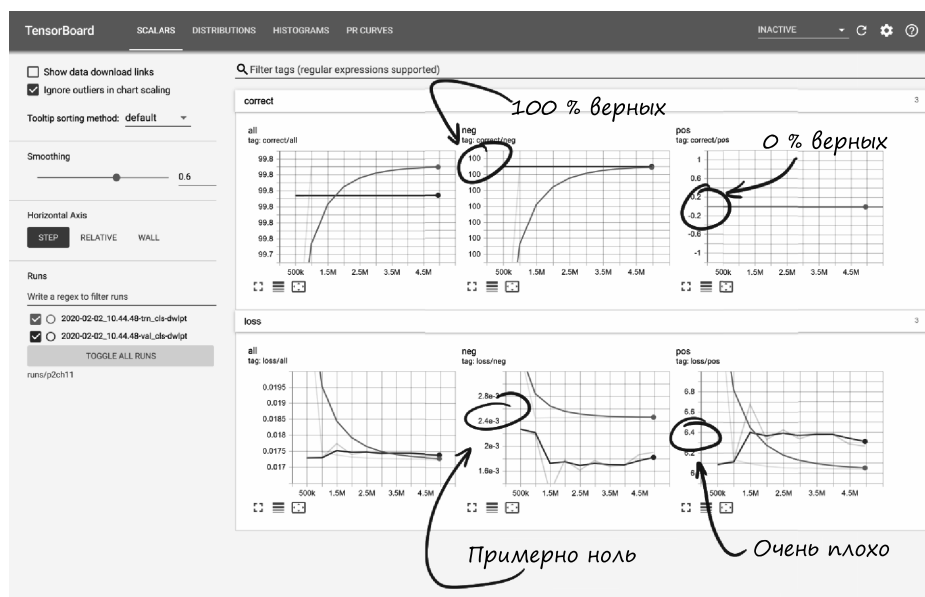


Рис. 11.12. В области отображения данных TensorBoard видно, что наши результаты по реальным узлам просто ужасны

Графики намного легче анализировать и усваивать, чем сообщения вида `E1 trn_pos 924,34, correct 0,2% (3 of 1215)`! Смысл содержимого этих графиков мы обсудим в разделе 11.10, но сейчас нужно обрести точное понимание того, как выведенные во время работы числа соотносятся с графиками. Попробуйте самостоятельно сопоставить числа, выданные сценарием `training.py`, со значениями на графиках. Вы должны увидеть прямое соответствие между столбцом `Value` во всплывающей подсказке и значениями, напечатанными во время обучения. Когда вы точно поймете, какие числа показывает `TensorBoard`, мы можем продолжить и обсудить, как сделать так, чтобы эти цифры отображались в первую очередь.

11.9.2. Внедрение `TensorBoard` в функцию регистрации метрик

Мы будем использовать модуль `torch.utils.tensorboard` для записи данных в формате, который будет применять `TensorBoard`. Это позволит нам быстро и легко писать метрики для этого и любого другого проекта. Инструмент `TensorBoard` поддерживает работу с массивами `NumPy` и тензорами `PyTorch`, но, поскольку у нас нет причин помещать наши данные в массивы `NumPy`, мы будем использовать исключительно тензоры `PyTorch`.

В первую очередь нам нужно создать объекты `SummaryWriter` (которые мы импортировали из `torch.utils.tensorboard`). В качестве единственного параметра передадим `log_dir`, присвоив ему значение вроде `runs/p2ch11/2020-01-01_12.55.27-trn-dlwpt`. Мы можем добавить в сценарий обучения аргумент комментария, чтобы вместо `dlwpt` выводилось нечто более информативное. Подробнее: `python -m p2ch11.training -help`.

Мы создаем два модуля записи, по одному для циклов обучения и проверки. Эти модули будут повторно использоваться в каждой эпохе. Когда класс `SummaryWriter` инициализируется, он также создает каталоги `log_dir`. Они отображаются в `TensorBoard` и могут загромождать пользовательский интерфейс пустыми запусками, если обучающий сценарий ломается до момента записи каких-либо данных. Это часто происходит, когда вы с чем-то экспериментируете.

Чтобы избежать записи слишком большого количества пустых запусков, мы откладываем создание экземпляров объектов `SummaryWriter` до тех пор, пока не будем готовы к первой записи. Эта функция вызывается из `logMetrics()` (листинг 11.21).

Листинг 11.21. `training.py:127, .initTensorboardWriters`

```
def initTensorboardWriters(self):
    if self.trn_writer is None:
        log_dir = os.path.join('runs', self.cli_args.tb_prefix, self.time_str)
```

```

self.trn_writer = SummaryWriter(
    log_dir=log_dir + '-trn_cls-' + self.cli_args.comment)
self.val_writer = SummaryWriter(
    log_dir=log_dir + '-val_cls-' + self.cli_args.comment)

```

Если помните, в эпохе 1 ничего годного достичь не удалось, а ранний вывод в цикле обучения был, по существу, случайным. Когда мы сохраняем метрики из первой партии, случайные результаты в конечном итоге немного искажают ситуацию. Вспомните рис. 11.11 и то, что в TensorBoard есть функция сглаживания для удаления шума из линий тренда, которая отчасти помогает справиться с этой проблемой.

Еще один подход — полностью пропустить метрики для обучающих данных эпохи 1, но если модель обучается достаточно быстро, то бывает полезно увидеть данные эпохи 1. Вы можете редактировать это поведение по своему усмотрению. В книге же на протяжении всей части II эпоха 1 будет включаться в данные.

СОВЕТ

Если в конечном итоге вы проведете много экспериментов, которые приведут к возникновению исключений или относительно быстрому уничтожению обучающего сценария, то у вас может остаться несколько ненужных запусков, загромождающих ваш каталог `run/`. Смело удаляйте их!

Запись скаляров в TensorBoard

Запись скаляров выполняется просто. Мы можем взять `metrics_dict`, который уже построили, и передавать каждую пару «ключ — значение» в метод `writer.add_scalar`. У класса `torch.utils.tensorboard.SummaryWriter` есть метод `add_scalar` (<http://mng.bz/RAqj>) с подписью, показанной в листинге 11.22.

Листинг 11.22. PyTorch `torch/utils/tensorboard/writer.py:267`

```

def add_scalar(self, tag, scalar_value, global_step=None, walltime=None):
    # ...

```

Параметр `tag` сообщает TensorBoard, на какой график мы добавляем значения, а параметр `scalar_value` — это значение оси Y нашей точки данных. Параметр `global_step` действует как значение оси X.

Напомним, что мы обновили переменную `totalTrainingSamples_count` внутри функции `doTraining`. Значение `totalTrainingSamples_count` будет отображаться на оси X нашего графика TensorBoard через параметр `global_step`. Вот как это выглядит в нашем коде (листинг 11.23).

Листинг 11.23. training.py:323, LunaTrainingApp.logMetrics

```
for key, value in metrics_dict.items():
    writer.add_scalar(key, value, self.totalTrainingSamples_count)
```

Обратите внимание, что косая черта в именах ключей (например, 'loss/all') позволяет TensorBoard группировать диаграммы по подстроке перед '/ '.

Документация предполагает, что в качестве параметра `global_step` мы должны передавать номер эпохи, но это приводит к некоторым сложностям. Используя число точек данных, переданных сети, мы можем изменять число точек данных на эпоху, не теряя возможности сравнивать эти будущие графики с теми, которые мы создаем сейчас. Бессмысленно говорить, что модель обучается за половину эпох, если каждая эпоха занимает в четыре раза больше времени! Однако имейте в виду, что это довольно нестандартная практика и на оси глобального шага окажется множество значений.

11.10. ПОЧЕМУ МОДЕЛЬ НЕ УЧИТСЯ ОБНАРУЖИВАТЬ УЗЕЛКИ?

Модель явно *чему-то* учится — убывающий тренд виден постоянно по мере увеличения эпох, а результаты воспроизводимы. Однако модель изучает что-то свое, а не то, что мы *хотели бы*. Что происходит? Проиллюстрируем проблему с помощью быстрой метафоры.

Представьте, что некий профессор дает студентам тест, состоящий из 100 вопросов вида «верно/неверно». Студенты видели предыдущие версии тестов этого профессора 30-летней давности, и в них всегда был *один или два* вопроса с ответом «верно». Остальные 98 или 99 — всегда «неверно».

Предположим, что оценки за тест выставляются стандартным методом, то есть 90 % правильных ответов соответствуют оценке 5. Тогда получить 5+ проще простого: просто отвечайте «неверно» на все вопросы! Представим, что в этом году в тесте только один ответ «верно» (рис. 11.13). Студент слева, который бездумно отвечал «неверно» на все вопросы, получил бы 99 % правильных ответов, но это не значит, что он чему-то научился (кроме как подглядывать в старые тесты). Примерно этим и занимается наша модель.

Теперь рассмотрим студента справа, который также ответил правильно на 99 % вопросов, ответив на два вопроса «верно». Интуиция подсказывает нам, что студент справа, вероятно, гораздо лучше знает материал, чем левый. Довольно сложно найти из 100 вопросов тот, на который ответ «верно»! К сожалению, ни

оценки наших учеников, ни схема оценивания нашей модели не отражают этого внутреннего ощущения.

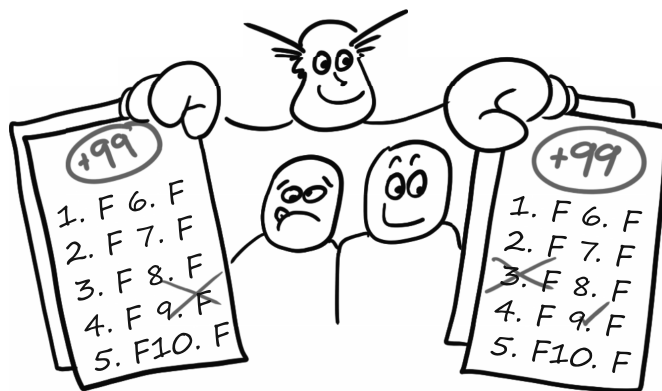


Рис. 11.13. Профессор ставит двум студентам одинаковые оценки, несмотря на разный уровень знаний. Вопрос 9 — единственный вопрос с ответом «верно»

У нас аналогичная ситуация, когда 99,7 % ответов на вопрос «Данный кандидат — узелок?» — это «нет». Наша модель выбирает легкий путь и отвечает False на каждый запрос.

Однако если мы более внимательно посмотрим на цифры нашей модели, то увидим, что потери в обучающих и проверочных наборах *уменьшатся*! Хотя какие-то подвижки в решении проблемы обнаружения рака должны вселять в нас надежду. В главе 12 этот потенциал будет развит далее. В начале главы мы введем новые важные термины, а затем придумаем лучшую схему оценивания, которую не так легко обмануть, как нынешнюю.

11.11. ИТОГИ ГЛАВЫ

В этой главе мы прошли долгий путь — теперь у нас есть и модель, и цикл обучения, и мы научились использовать данные, полученные в предыдущей главе. Наши метрики записываются в консоль, а также отображаются в виде графика.

Эти результаты пока непригодны для использования, однако на самом деле мы ближе к цели, чем может показаться. В главе 12 мы улучшим метрики, которые используются для отслеживания прогресса, и с их помощью научимся понимать, какие изменения необходимо внести, чтобы наша модель давала разумные результаты.

11.12. УПРАЖНЕНИЯ

1. Реализуйте программу, которая итерирует экземпляр `LunaDataset`, заключая его в экземпляр `DataLoader`, определяя при этом время, необходимое на выполнение задачи. Сравните это время со временем из упражнений в главе 10. Следите за состоянием кэша при выполнении скрипта.
 - А. Какое влияние оказывает настройка `num_workers=0, 1 и 2`?
 - Б. Какие максимальные значения будет поддерживать ваша машина для данной комбинации `batch_size=...` и `num_workers=...`, чтобы не закончилась память?
2. Отсортируйте `noduleInfo_list` в обратном направлении. Как это меняет поведение модели после одной эпохи обучения?
3. Измените `logMetrics`, чтобы изменить схему именования запусков и ключей, используемых в `TensorBoard`.
 - А. Поэкспериментируйте с различным размещением косой черты в ключах, передаваемых в `writer.add_scalar`.
 - Б. Используйте один и тот же модуль записи для циклов обучения и проверки и добавьте строку `trn` или `val` к имени ключа.
 - В. Настройте имя каталога для логов и ключей.

11.13. РЕЗЮМЕ

- Загрузчики данных можно использовать для загрузки данных из произвольных наборов в нескольких процессах. Это позволяет направить бездействующие ресурсы ЦП на подготовку данных для ГП.
- Загрузчики данных загружают несколько элементов из набора данных и объединяют их в пакет. Модели PyTorch предполагают обработку пакетов данных, а не отдельных элементов.
- Загрузчики данных можно использовать для управления произвольными наборами данных путем изменения относительной частоты отдельных элементов. Это позволяет вносить «послепродажные» изменения в набор данных, хотя может быть более целесообразным напрямую изменить реализацию набора данных.
- В части II мы в основном будем использовать оптимизатор PyTorch `torch.optim.SGD` (стохастический градиентный спуск) со скоростью обучения 0,001 и импульсом 0,99. Эти значения также являются разумными значениями по умолчанию для многих проектов глубокого обучения.

- Наша первоначальная модель классификации будет очень похожа на модель, которую мы использовали в главе 8. Это позволит нам начать с модели, которая, как мы полагаем, будет эффективной. Мы можем вернуться к дизайну модели, если считаем, что именно он мешает нашему проекту работать лучше.
- Большое значение имеет выбор метрик, которые мы отслеживаем во время обучения. Легко случайно выбрать метрики, неверно показывающие работу модели. Использовать общий процент правильно классифицированных образцов для наших данных бессмысленно. В главе 12 подробно рассказывается, как оценивать и выбирать лучшие метрики.
- С помощью инструмента **TensorBoard** можно визуально отображать широкий спектр метрик. Это значительно упрощает использование определенных форм данных (особенно данных трендов), поскольку они меняются в зависимости от эпохи обучения.

Улучшение процесса обучения с помощью метрик и дополнений

В этой главе

- ✓ Определение и вычисление точности, отзыва и количества истинных/ложных положительных/отрицательных результатов.
- ✓ Сравнение метрики F1 с другими метриками качества.
- ✓ Балансировка и дополнение данных для уменьшения переобучения.
- ✓ Построение графиков метрик качества с помощью инструмента TensorBoard.

В конце предыдущей главы мы оказались в затруднительном положении. В целом мы сумели реализовать общий механизм для нашего проекта глубокого обучения, но ни один из результатов не оказался полезным, так как сеть просто классифицировала все данные как не узелки! Что еще хуже, на первый взгляд результаты выглядели отличными, поскольку большая часть обучающих и проверочных данных оказалась классифицирована правильно. Наши данные сильно смещены в сторону отрицательных элементов (не узелков), поэтому, назвав все данные не узелками, модель легко и просто получает высокую оценку. Жаль, что при этом она оказывается бесполезной!

Это означает следующее: на рис. 12.1 нас пока интересует та же часть схемы, что и в главе 11. Однако на сей раз нужно сделать так, чтобы наша классификационная модель работала *хорошо*, а не *хоть как-то*. Эта глава посвящена тому,

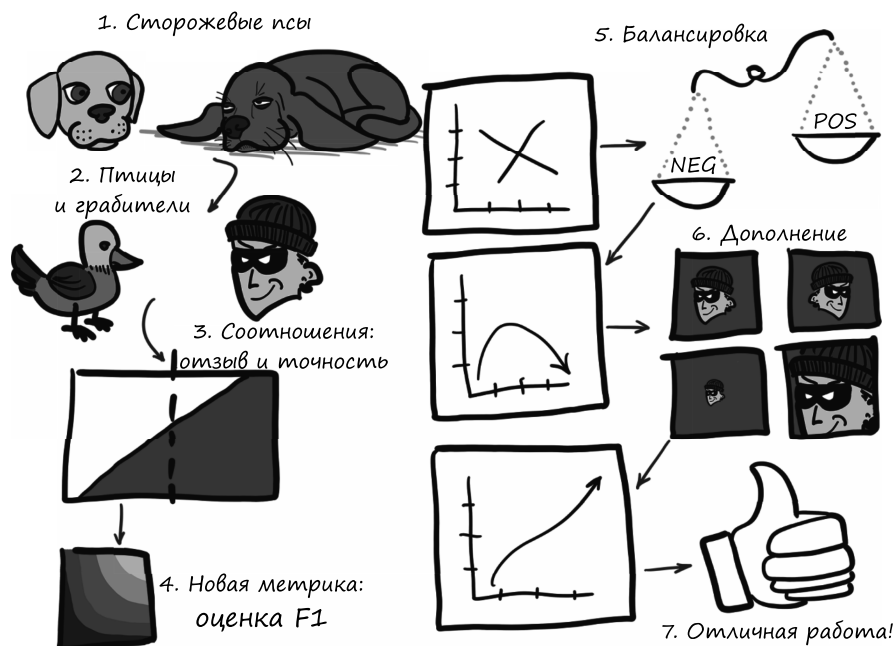


Рис. 12.2. Метафоры, которые мы будем использовать для изобретения новых прекрасных метрик для модели

Наконец, для улучшения результатов обучения мы внесем необходимые изменения в нашу реализацию `LunaDataset`: балансировка (5) и дополнение (6). Мы посмотрим, помогут ли эти экспериментальные изменения исправить показатели производительности. К концу главы наша обученная модель будет работать намного лучше: отличная работа (7)! Модель все еще не будет готова к клиническому использованию, но сможет давать результаты, явно лучшие, чем случайные. В конце главы мы получим рабочую реализацию этапа 4: классификации узелков-кандидатов. Закончив работу, мы можем начать думать о том, как включить в проект этапы 2 (сегментация) и 3 (группировка).

12.2. ХОРОШИЕ СОБАКИ ПРОТИВ ПЛОХИХ ПАРНЕЙ: ЛОЖНОПОЛОЖИТЕЛЬНЫЕ И ЛОЖНООТРИЦАТЕЛЬНЫЕ РЕЗУЛЬТАТЫ

Вместо моделей и опухолей рассмотрим двух сторожевых собак, изображенных на рис. 12.3 и только что прошедших обучение и дрессировку. Собаки обучены сообщать хозяину о появлении грабителя — редкая, но серьезная ситуация, требующая немедленного внимания.

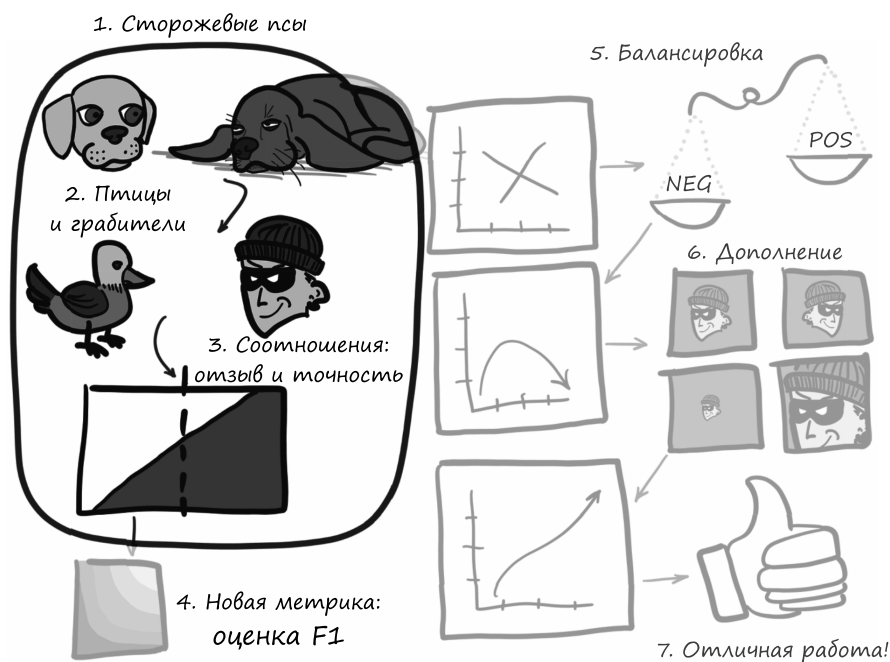


Рис. 12.3. Тема этой главы. Выделена используемая метафора с собаками

К сожалению, хотя обе собаки и хороши, ни одна из них не тянет на профессионального *сторожевого* пса. Терьер (Рокси) лает практически на все, а старый гончий пес (Престон) лает почти исключительно на грабителей, но только если он не спит, когда они приходят.

Рокси будет почти каждый раз сообщать нам о грабителе. А еще о пожарных машинах, грозах, вертолетах, птицах, почтальонах, белках, прохожих и обо всем остальном. Если мы будем откликаться на каждый лай, нас почти никогда не ограбят (и лишь хитрейший из воров сможет проскользнуть). Идеально!.. За исключением того, что такое усердие сторожевой собаки сводит на нет всякую экономию наших усилий. Нам придется вылезать из кровати каждые пару часов с фонариком в руке, поскольку Рокси учуяла кошку, или услышала сову, или увидела, как мимо проезжает поздний автобус. У Рокси слишком много ложноположительных результатов.

Ложноположительный результат — событие, которое нас теоретически интересует или относится к желаемому классу (то есть именно то, что мы пытаемся обнаружить), но при ближайшем рассмотрении оказывается *неважным*. В задаче с поиском узелков это случай, когда фактически неинтересный кандидат отмечается как узелок и, следовательно, нуждается во внимании радиолога. Для Рокси это пожарные машины, кошки и т. д. В качестве канонического примера

ложноположительного результата в следующем разделе и в последующей главе мы будем использовать изображение кошки.

Сравним ложноположительный результат с *истинно положительным*: это случай, когда интересующее нас событие классифицировано правильно. На рисунках оно будет представлено грабителем-человеком.

Вернемся ко второй собаке: если Престон лает, то стоит вызывать полицию, поскольку это почти наверняка означает, что кто-то вломился в дом, начался пожар или на город напала Годзилла. Однако Престон крепко спит, и звук вторжения в дом вряд ли разбудит его, поэтому нас все равно будет грабить каждый встречный. Такой результат лучше, чем ничего, но не позволяет достичь того душевного спокойствия, ради которого мы, собственно, изначально и завели собаку. У Престона слишком много ложноотрицательных результатов.

Ложноотрицательный результат — событие, которое классифицируется системой как не представляющее интереса или не относящееся к желаемому классу, однако на самом деле является важным. В задаче с обнаружением узелков это случай, когда узелок (то есть потенциальный рак) остается незамеченным. У Престона это грабежи, которые он проспал. Для иллюстрации ложноотрицательных результатов мы проявим немного креатива и изобразим грабителя-грызуна!

Сравним ложноотрицательные результаты с *истинно отрицательными*: это неинтересные события, которые классифицированы правильно. Для примера возьмем изображение птицы.

Подведем итог метафоры. Модель, составленная в главе 11, — это кошка, которая отказывается мяукать на что-либо, кроме банки с тунцом (при этом стоически игнорируя Рокси). В конце предыдущей главы мы вычисляли процент правильных ответов для всего обучающего и проверочного наборов. Стало ясно, что это был не лучший способ оценить работу модели. Чрезмерное внимание каждой из наших собак к одному показателю (например, количеству истинно положительных или истинно отрицательных) показало, что для правильной оценки общей эффективности нам нужна метрика с более широким охватом.

12.3. ВИЗУАЛИЗАЦИЯ ПОЛОЖИТЕЛЬНЫХ И ОТРИЦАТЕЛЬНЫХ РЕЗУЛЬТАТОВ

Приступим к разработке визуального языка, который мы будем использовать для описания истинных/ложных положительных/отрицательных результатов. Пожалуйста, потерпите, если наше объяснение будет повторяться. Мы хотим точно убедиться, что вы правильно сформировали мысленную модель коэффициентов, которые мы введем далее. Рассмотрим рис. 12.4: на нем показаны события, которые могут представлять интерес для одной из наших сторожевых собак.



Рис. 12.4. Кошки, птицы, грызуны и грабители составляют четыре квадранта нашей классификации. Квадранты разделены порогами обнаружения с точки зрения человека и собак

На рис. 12.4 изображены два порога. Первый — установленная людьми разделятельная линия, отделяющая грабителей от безобидных животных. В нашем случае это метка, которая присваивается каждому элементу данных из обучающего или проверочного набора. Вторым — определяемый собакой *порог классификации*, показывающий, на что реагирует собака. Для модели глубокого обучения это прогнозируемое значение, которое модель выдает при рассмотрении элемента данных. Сочетание этих двух порогов делит наши события на квадранты: истинные/ложные положительные/отрицательные. Закрасим тревожные события более темным цветом (поскольку злодеи всегда крадутся в темноте).

Конечно, реальность устроена гораздо сложнее. Не существует ни платоновского идеала грабителя, ни единого для всех классификационного порога, ниже которого будут находиться все грабители. На рис. 12.5 показано, что одни грабители будут хитрее других, а некоторые птицы — надоедливее. Продолжая тему, мы заключим наши примеры в график. По оси *X* откладывается ценность лая собаки для данного события. Ось *Y* отражает некий неопределенный набор качеств, которые мы, люди, можем воспринимать, а собаки — нет.

Поскольку наша модель производит бинарную классификацию, порог прогнозирования можно промоделировать с помощью сравнения выходных данных, приведенных к одному числовому значению, с пороговым значением

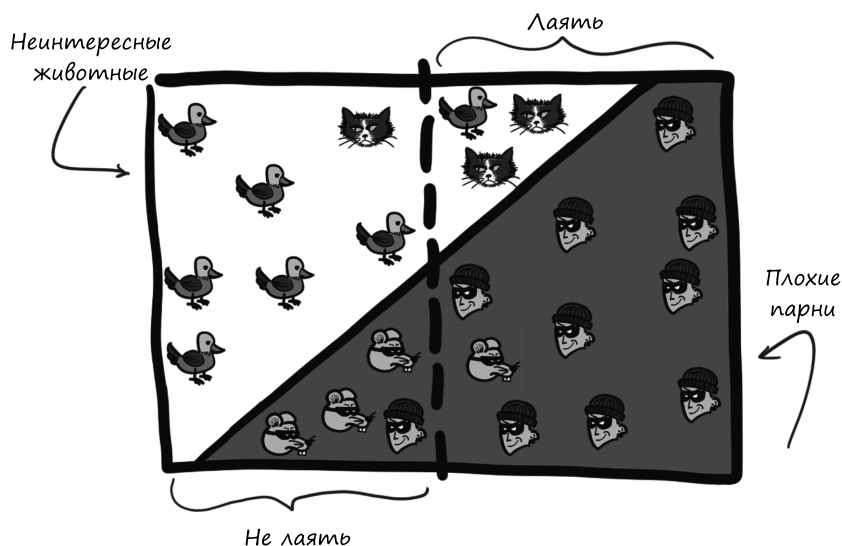


Рис. 12.5. Для каждого типа события есть множество возможных случаев, которые собаки должны будут оценить

классификации. Именно поэтому мы потребуем, чтобы линия порога классификации на рис. 12.5 была строго вертикальной.

Каждый взломщик индивидуален, поэтому нашим сторожевым собакам придется оценивать множество различных ситуаций, а это означает больше шансов ошибиться. Мы можем провести четкую диагональную линию, отделяющую птиц от грабителей, но Престон и Рокси различают здесь только ось X , и из-за этого в центральной части графика события перемешиваются. Собаки должны выбрать вертикальный порог, но это означает, что ни одна собака не сможет отработать идеально. Иногда человек, который тащит вашу технику в свой фургон, — просто мастер по ремонту, которого вы сами же и наняли, а иногда в фургоне с надписью «Ремонт стиральных машин» сидят настоящие грабители. Наивно полагать, что собака сумеет различить эти нюансы.

У фактических входных данных, которые мы будем использовать, весьма высокая размерность: нам нужно рассмотреть тысячи вокселей КТ, а также более абстрактные вещи, такие как размер кандидата, его расположение в легких и т. д. Задача нашей модели состоит в том, чтобы отобразить каждое из этих событий и их свойств в данный прямоугольник таким образом, чтобы мы сумели четко разделить положительные и отрицательные события, проведя одну вертикальную линию (то есть порог классификации). Эту работу будут выполнять слои `nn.Linear` в конце модели. Положение вертикальной линии точно соответствует значению `classificationThreshold_float`, которое мы видели в подразделе 11.6.1. Но там мы просто жестко задали пороговое значение 0,5.

Обратите внимание, что на самом деле представленные данные не являются двумерными. После предпоследнего слоя они превращаются из многомерных в одномерные (ось X), и в итоге мы получаем всего один скаляр для каждого элемента данных (который затем делится пополам порогом классификации). Далее мы используем второе измерение (ось Y) для представления характеристик каждого элемента данных, которые наша модель не в состоянии видеть или применить: это могут быть возраст или пол пациента, расположение кандидата в легких или даже локальные свойства кандидатов, которые модель вообще не использовала. Таким образом мы можем представить ту самую путаницу между настоящими и ненастоящими узелками.

Области квадрантов на рис. 12.5 и количество выборок, которые в них попадут, понадобятся для вычисления эффективности модели. Мы можем использовать отношения между этими значениями для создания более сложных метрик, которые гораздо лучше подходят для объективного измерения качества работы. Как говорится, «доказательство находится в пропорциях»¹. Затем мы возьмем соотношения между этими подмножествами событий, чтобы определить нужные метрики.

12.3.1. Высокий отклик Рокси

Отклик — показатель того, что «мы никогда не пропустим ни одного интересного события!». Формально *отклик* — это отношение истинно положительных результатов к сумме истинно положительных и ложноотрицательных результатов. Проиллюстрируем это на рис. 12.6.

ПРИМЕЧАНИЕ

Отклик также называют чувствительностью.

Чтобы улучшить отклик, нужно свести к минимуму ложноотрицательные результаты. В терминах сторожевых собак это означает, что в любой непонятной ситуации лучше лаять. Не в мою смену, господа грабители!

У Рокси невероятно высокий уровень отклика, и ее классификационный порог сдвинут до упора влево, охватывая почти все положительные события на рис. 12.7. В данном случае ее значение отклика близко к 1,0, то есть 99 % грабителей будут обнаружены. Поскольку именно так Рокси определяет успех, по ее мнению, она отлично справляется со своей задачей. А огромное количество ложных срабатываний — это так, ерунда!

¹ На самом деле этого никто не говорит.

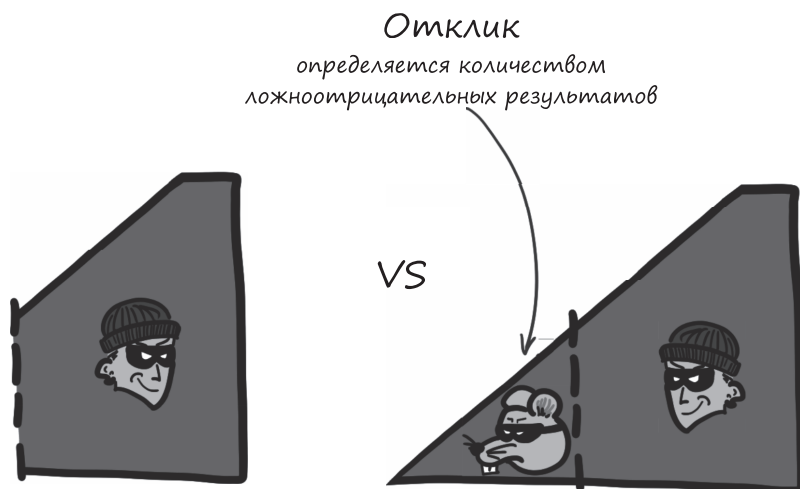


Рис. 12.6. Отклик — это отношение истинно положительных результатов к сумме истинно положительных и ложноотрицательных результатов. Высокий отклик сводит ложноотрицательные результаты к минимуму

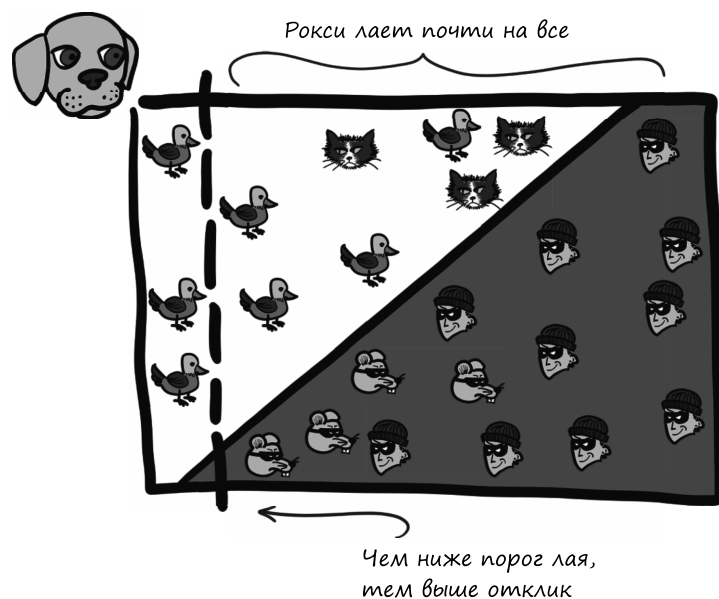


Рис. 12.7. Порог, выбранный Рокси, сводит к минимуму число ложноотрицательных результатов. Она лает и на кошек, и на крыс, и почти на всех птиц

12.3.2. Высокая точность Престона

Точность — это правило «Никогда не лай, если не уверен». Чтобы повысить точность, нужно свести к минимуму ложноположительные результаты. Престон не будет лаять на кого-то, не будучи уверенным, что это грабитель. Если говорить более формально, *точность* — это отношение истинно положительных результатов к сумме истинно положительных и ложноположительных, как показано на рис. 12.8.

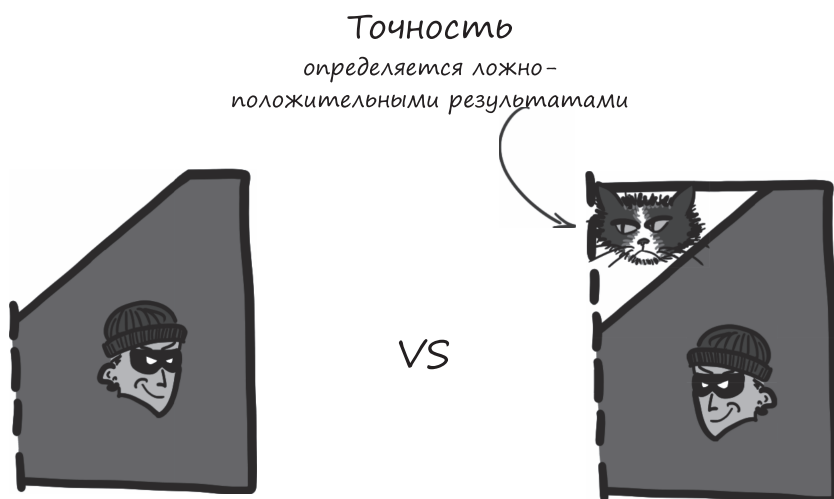


Рис. 12.8. Точность — это отношение истинно положительных результатов к сумме истинно положительных и ложноположительных. Высокая точность сводит к минимуму ложноположительные результаты

У Престона невероятно высокая точность, и его порог классификации сдвинут до упора вправо, поэтому он отсеивает как можно больше неинтересных негативных событий (рис. 12.9). Его подход противоположен подходу Рокси и означает, что у Престона точность близка к 1,0: 99 % тех, на кого он лает, — грабители. Это тоже соответствует его определению хорошей сторожевой собаки, даже несмотря на то, что большое количество событий проходит незамеченными.

Хотя ни точность, ни отклик не могут выступать в роли единственной метрики, используемой для оценки модели, оба эти показателя полезны, и их нужно иметь под рукой во время обучения. Начнем вычислять и выводить их в процессе обучения, а затем обсудим другие показатели, которые можно будет применить.



Рис. 12.9. Порог Престона минимизирует число ложных срабатываний. Кошек никто не трогает, интересны лишь грабители!

12.3.3. Реализация точности и отклика в `logMetrics`

Точность и отклик — это ценные показатели, которые можно отслеживать во время обучения, поскольку они дают важную информацию о поведении модели. Если любой из них упадет до нуля (как мы видели в главе 11!), то стоит предположить, что наша модель начала вести себя вырожденным образом. Имея под рукой точные детали поведения, мы можем понять, что нужно исследовать и с чем экспериментировать, чтобы вернуть обучение в нужное русло. Нам нужно обновить функцию `logMetrics`, чтобы научить ее выводить значения точности и отклика в выходные данные каждой эпохи, в дополнение к значениям потерь и корректности.

Мы уже определили точность и полноту в терминах истинно положительных результатов и т. п., поэтому в коде поступим так же. Оказывается, у нас уже есть некоторые нужные нам значения, хотя они и названы по-другому (листинг 12.1).

Листинг 12.1. `training.py:315, LunaTrainingApp.logMetrics`

```
neg_count = int(negLabel_mask.sum())
pos_count = int(posLabel_mask.sum())

trueNeg_count = neg_correct = int((negLabel_mask & negPred_mask).sum())
truePos_count = pos_correct = int((posLabel_mask & posPred_mask).sum())
```

```
falsePos_count = neg_count - neg_correct
falseNeg_count = pos_count - pos_correct
```

Здесь мы видим, что `neg_correct` — то же самое, что и `trueNeg_count`! Неудивительно, так как отсутствие узелка — это «отрицательный» результат (как в «отрицательном диагнозе»), и если классификатор дает правильный прогноз, то это истинно отрицательный результат. Правильно маркированные образцы узелков являются истинно положительными.

Осталось добавить переменные для наших ложноположительных и ложноотрицательных значений. Это несложно, поскольку будет достаточно взять общее количество доброкачественных образований и вычесть из них количество правильно определенных. Теперь подсчитаем точки данных, в которых нет узелков, но которые ошибочно классифицированы как *положительные*. Это будут ложноположительные результаты. Опять же, расчет будет выполняться по тому же признаку, но использоваться будет количество узелков.

Имея эти значения, мы можем вычислить параметры `precision` и `recall` и сохранить их в `metrics_dict` (листинг 12.2).

Листинг 12.2. training.py:333, LunaTrainingApp.logMetrics

```
precision = metrics_dict['pr/precision'] = \
    truePos_count / np.float32(truePos_count + falsePos_count)
recall = metrics_dict['pr/recall'] = \
    truePos_count / np.float32(truePos_count + falseNeg_count)
```

Обратите внимание на двойное присваивание: использовать отдельные переменные `precision` и `recall` не столь уж необходимо, но так читабельность следующего раздела улучшится. Вдобавок мы расширяем оператор `logging` в `logMetrics`, чтобы включить в него новые значения, но пока опускаем реализацию (вернемся к логированию позже в этой главе).

12.3.4. Готовая метрика производительности: метрика F1

Ни точность, ни полнота не отражают в полной мере то, что нам нужно для объективной оценки модели, хотя это и полезные показатели. Как мы видели на примере Рокси и Престона, каждый из этих параметров можно рассматривать по отдельности, сдвигая порог классификации. В результате получается модель, которая хороша в цифрах, но мало полезна в реальном мире. Нам нужно что-то сочетающее в себе оба этих значения таким образом, чтобы предотвратить неправильное поведение. Как видно из рис. 12.10, пришло время представить окончательную метрику. Общепринятый способ сочетания точности и полноты — использование метрики F1 (https://en.wikipedia.org/wiki/F1_score). Как и другие оценочные показатели, метрика F1 находится в диапазоне от 0 (классификатор без реальной предсказательной силы) до 1 (классификатор,

дающий идеальные прогнозы). Мы также научим `logMetrics` выводить эту метрику (листинг 12.3).

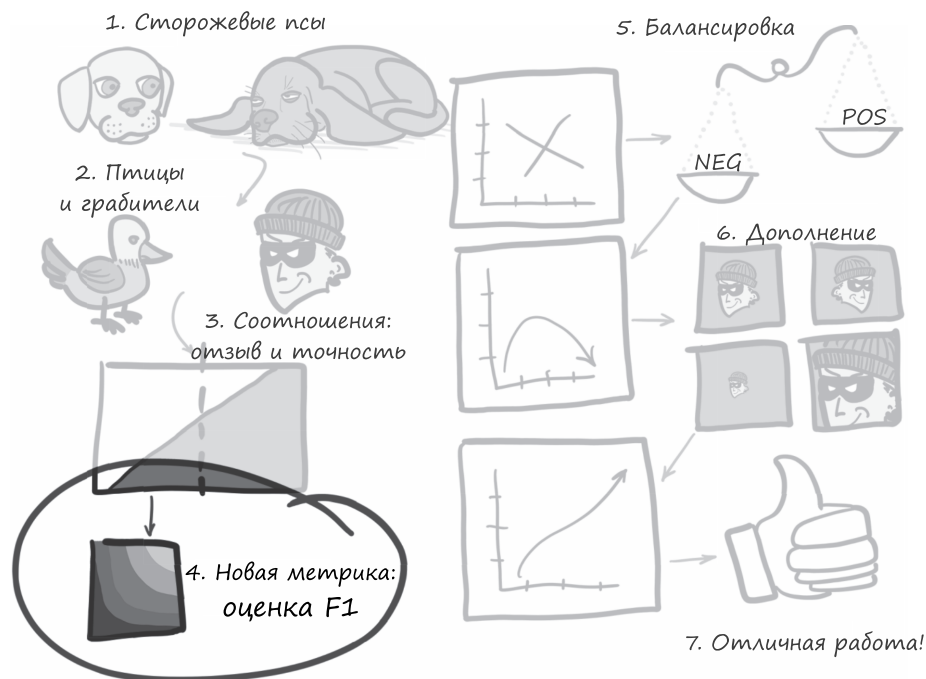


Рис. 12.10. Темы этой главы. Выделен этап определения метрики

Листинг 12.3. `training.py:338, LunaTrainingApp.logMetrics`

```
metrics_dict['pr/f1_score'] = \
    2 * (precision * recall) / (precision + recall)
```

На первый взгляд метрика кажется более сложной, чем нам нужно, и может быть не сразу очевидно, как она поведет себя, когда придется жертвовать точностью ради отклика и наоборот.

Однако эта формула обладает множеством замечательных свойств и выгодно отличается от нескольких других, более простых альтернатив, которые мы могли бы рассмотреть.

Как вариант, мы могли бы просто усреднить значения точности и отклика. К сожалению, в таком случае $\text{avg}(p=1.0, r=0.0)$ и $\text{avg}(p=0.5, r=0.5)$ дает оценку 0,5, а мы уже знаем, что классификатор с нулевой точностью или откликом обычно не имеет ценности. Присвоение чему-то бесполезному отличного от нуля значения, словно чему-то полезному, сразу дает понять, что такая метрика ни на что не годится.

Наглядно сравним усреднение и F1 на рис. 12.11. Сразу выделяются несколько вещей. Для начала, у усреднения нет кривой или изгиба в линиях. Именно это позволяет точности или отклику искажаться в одну или другую сторону! *Не бывает* ситуаций, когда не имело бы смысла максимизировать оценку за счет 100 % отклика (подход Рокси), а затем устранить любые ложные срабатывания. При этом мы получаем оценку работы 0,5 по щелчку пальцев! Такой подход кажется неправильным.

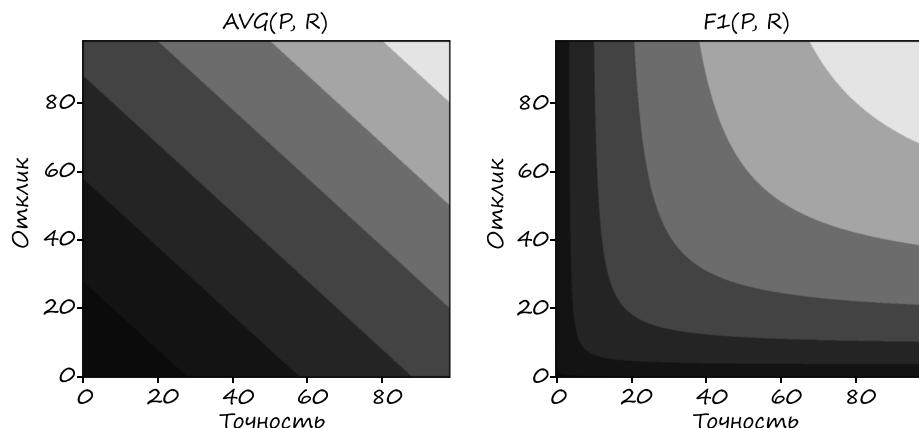


Рис. 12.11. Вычисление окончательной оценки с помощью $\text{avg}(p, r)$. Более светлые значения ближе к 1,0

ПРИМЕЧАНИЕ

На самом деле здесь мы берем среднее арифметическое (https://en.wikipedia.org/wiki/Arithmetic_mean) точности и отклика, а оба эти значения являются отношениями, а не скалярными значениями. Взятие среднего арифметического отношений обычно не дает значимых результатов. Метрика F1 — это другое название среднего гармонического (https://en.wikipedia.org/wiki/Harmonic_mean) из двух отношений, которое лучше подходит для объединения подобных значений.

Сравним усреднение с метрикой F1: при высоком отклике и низкой точности даже небольшой обмен одного показателя на другой приблизит оценку к точке наилучшего результата. На графике виден хороший, глубокий изгиб, в который легко скользить. Поощрение баланса точности и отзыва — именно то, что нам нужно.

Допустим, нам нужна более простая метрика, которая вообще против асимметрии. Чтобы скорректировать слабость сложения, мы могли бы взять минимум точности и полноты (рис. 12.12).

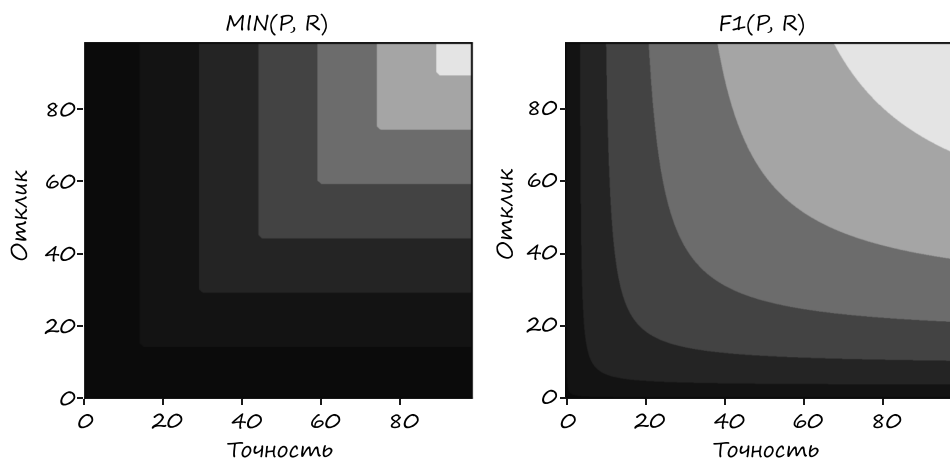


Рис. 12.12. Вычисление окончательной оценки как $\min(p, r)$

Данная метрика неплоха, ведь если одно из значений равно 0, то оценка также равна 0, и единственный способ получить оценку 1,0 — добиться, чтобы оба значения были равны 1,0. Но и эта метрика оставляет желать лучшего, поскольку внесение в модель изменений, которые бы увеличили отклик с 0,7 до 0,9, оставив неизменной точность на уровне 0,5, никак не влияет на оценку, равно как и снижение отклика до 0,6! Конечно, эта метрика показывает дисбаланс между точностью и откликом, но не отражает многих нюансов этих двух значений. Как мы видели, легко разменять одно на другое, просто изменив порог классификации. Мы хотели бы, чтобы наша метрика отражала такие сделки.

Поэтому для достижения цели все придется немного усложнить. Мы могли бы перемножить эти значения, как показано на рис. 12.13. У данного подхода есть хорошее свойство: если один из показателей равен 0, то оценка тоже равна 0, а оценка 1,0 означает, что оба показателя идеальны. Вдобавок мы получаем сбалансированный компромисс между точностью и полнотой при низких значениях, а по мере приближения к идеальным результатам график становится более линейным. Это не очень хорошо, поскольку в данном случае для улучшения метрики придется сильно подтолкнуть оба показателя.

ПРИМЕЧАНИЕ

Здесь мы берем среднее геометрическое (https://en.wikipedia.org/wiki/Geometric_mean) двух отношений, что также не дает значимых результатов.

Есть и еще одна проблема: почти весь квадрант от (0, 0) до (0,5, 0,5) очень близок к нулю. Позже мы увидим, что важно иметь метрику, чувствительную к изменениям в этом регионе, особенно на ранних этапах проектирования нашей модели.

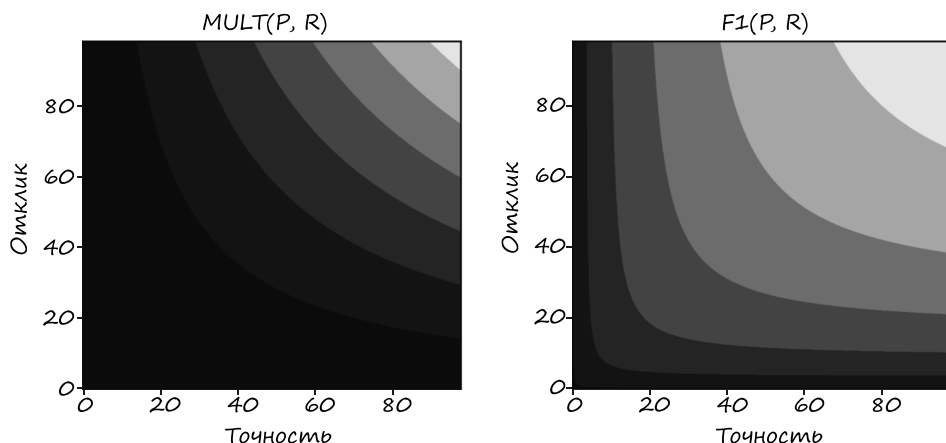


Рис. 12.13. Вычисление окончательной оценки с помощью функции `mult(p, r)`

В принципе, умножение тоже можно применять в качестве оценочной функции (которая не обладает недостатками предыдущего использованного варианта); в будущем оценивать производительность нашей модели классификации мы будем с помощью метрики F1.

Добавление точности, отклика и метрики F1 в выходные данные

Теперь, когда у нас есть новые метрики, добавить их в выходные данные журнала довольно просто. Добавим точность, отклик и F1 в функцию логирования для обучающих и проверочных данных (листинг 12.4).

Листинг 12.4. `training.py:341, LunaTrainingApp.logMetrics`

```
log.info(
    ("E{} {:.8} {loss/all:.4f} loss, "
     + "{correct/all:-5.1f}% correct, "
     + "{pr/precision:.4f} precision, "
     + "{pr/recall:.4f} recall, "
     + "{pr/f1_score:.4f} f1 score"
    ).format(
        epoch_ndx,
        mode_str,
        **metrics_dict,
    )
)
```

Обновление строки
формата

Кроме того, добавим в вывод точные значения количества правильно идентифицированных элементов и общее количество отрицательных и положительных элементов (листинг 12.5).

Листинг 12.5. training.py:353, LunaTrainingApp.logMetrics

```
log.info(
    ("E{ } { :8} {loss/neg:.4f} loss, "
     + "{correct/neg:-5.1f}% correct ({neg_correct:} of {neg_count:})")
    ).format(
        epoch_ndx,
        mode_str + '_neg',
        neg_correct=neg_correct,
        neg_count=neg_count,
        **metrics_dict,
    )
)
```

Новая версия положительного отчета о регистрации выглядит почти так же.

12.3.5. Как модель работает с новыми метриками

Теперь, когда мы внедрили классные новые метрики, попробуем их в деле. Результаты обсудим после того, как покажем результаты сеанса оболочки Bash. Возможно, вы захотите посмотреть результаты заранее, пока ваша система обрабатывает числа, а обработка может занять около получаса, в зависимости от вашей системы¹. Точное время будет зависеть от скорости вашего процессора, графического процессора и диска. Нашей системе с SSD и GTX 1080 Ti на полную эпоху потребовалось около 20 минут:

```
$ ../venv/bin/python -m p2ch12.training
```

```
Starting LunaTrainingApp...
```

```
...
```

```
E1 LunaTrainingApp
```

```
.../p2ch12/training.py:274: RuntimeWarning:
```

```
⇒ invalid value encountered in double_scalars
```

```
metrics_dict['pr/f1_score'] = 2 * (precision * recall) /
⇒ (precision + recall)
```

Точное количество и номера строк предупреждений RuntimeWarning могут различаться от запуска к запуску

```
E1 trn      0.0025 loss, 99.8% correct, 0.0000 prc, 0.0000 rcl, nan f1
```

```
E1 trn_ben  0.0000 loss, 100.0% correct (494735 of 494743)
```

```
E1 trn_mal  1.0000 loss,  0.0% correct (0 of 1215)
```

```
.../p2ch12/training.py:269: RuntimeWarning:
```

```
⇒ invalid value encountered in long_scalars
```

```
precision = metrics_dict['pr/precision'] = truePos_count /
⇒ (truePos_count + falsePos_count)
```

```
E1 val      0.0025 loss, 99.8% correct, nan prc, 0.0000 rcl, nan f1
```

```
E1 val_ben  0.0000 loss, 100.0% correct (54971 of 54971)
```

```
E1 val_mal  1.0000 loss,  0.0% correct (0 of 136)
```

¹ Если времени уходит больше, то проверьте, запустили ли вы сценарий prercache.

Приплыли. Мы получили несколько предупреждений, и, учитывая, что некоторые из вычисленных нами значений оказались `nan`, вероятно, где-то происходит деление на ноль. Посмотрим, что и где произошло.

Во-первых, поскольку *ни один* из положительных элементов в обучающем наборе не был классифицирован как положительный, это означает, что и точность, и отклик равны нулю, и поэтому при расчете метрики F1 получается деление на ноль. Во-вторых, у проверочного набора значения `truePos_count` и `falsePos_count` равны нулю из-за того, что *ни один* элемент не оказался отмечен как положительный. Отсюда следует, что знаменатель расчета точности также равен нулю. Это понятно, так как здесь мы видим еще одно `RuntimeWarning`.

Несколько отрицательных элементов данных были классифицированы как положительные (494 735 из 494 743 были определены как отрицательные, и восемь элементов были классифицированы неправильно). Поначалу это может показаться странным, но вспомните, что мы собираем результаты обучения на протяжении всей эпохи, а не используем состояние модели в конце эпохи, как в случае проверки. Это значит, что в первой партии результаты могут быть буквально случайными. Неудивительно, что несколько элементов из этой первой партии оказались помечены как положительные.

ПРИМЕЧАНИЕ

Из-за случайной инициализации весовых коэффициентов сети и случайного упорядочения обучающих выборок поведение каждого отдельного прогона может отличаться от других. Часто бывает желательно добиться точно воспроизводимого поведения, но это выходит за рамки того, что мы пытаемся сделать в части II этой книги.

Получилось обидно. Переход на новые метрики привел к переходу с «отличная работа» на «ничего не работает, и то, если повезет», а если не повезет, то *польза и вовсе уходит в минус*. Мда.

Тем не менее в долгосрочной перспективе это хорошо. Мы еще в главе 11 знали, что производительность модели была отвратительной. Если бы наши метрики дали нам другой результат, то это указывало бы на их фундаментальный недостаток!

12.4. КАК ВЫГЛЯДИТ ИДЕАЛЬНЫЙ НАБОР ДАННЫХ

Вместо того чтобы рыдать из-за плачевного положения дел, лучше подумаем о том, чего мы на самом деле хотим от модели. На рис. 12.14 показано, что для правильного обучения модели сначала нужно сбалансировать данные. Создадим логические шаги, которые для этого нужны.

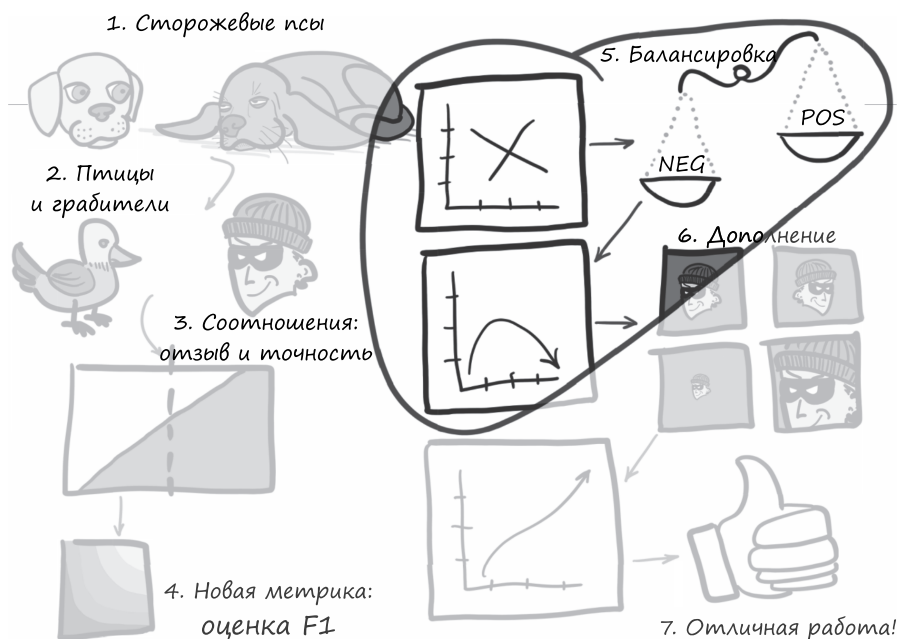


Рис. 12.14. Темы этой главы. Пора поговорить о балансе положительных и отрицательных элементов

Вспомните рис. 12.5 и наше обсуждение порогов классификации. Достижение лучших результатов за счет изменения порогового значения дает небольшие результаты, так как возникает слишком много совпадений между положительными и отрицательными классами, с которыми невозможно работать¹.

Вместо этого мы хотим получить изображение, подобное показанному на рис. 12.15. Порог у меток здесь почти вертикальный. Именно это нам и нужно, поскольку порог для меток и порог классификации должны совпадать в достаточно хорошей степени. Кроме того, большинство точек данных сконцентрировано по краям диаграммы. Чтобы достичь этого, данные должны быть легко разделимыми, а модель должна уметь делать это. Наша модель в данный момент достаточно сильна для этого, так что проблема не в ней. Вместо этого более пристально взглянем на данные.

Напомним, что данные чрезмерно несбалансированы. Соотношение положительных точек к отрицательным составляет 400:1. Это *катастрофически* большой дисбаланс! На рис. 12.16 показано, как это выглядит. Неудивительно, что настоящие узелки попросту теряются в общей массе!

¹ Имейте в виду, что эти изображения являются просто представлением классификационного пространства и не отражают истинное положение вещей.

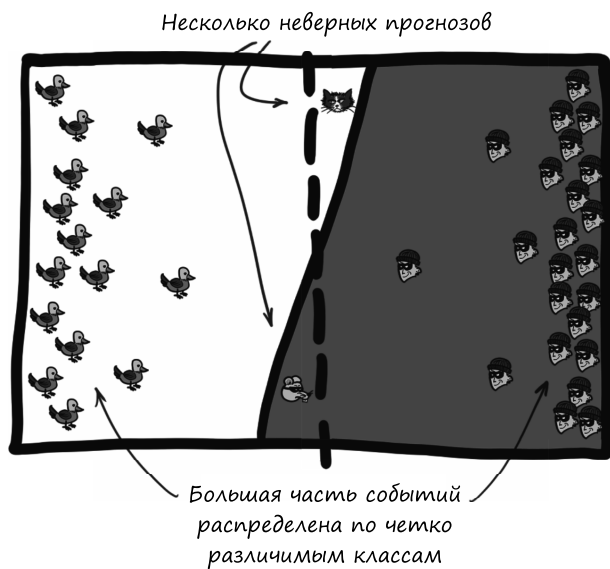


Рис. 12.15. Хорошо обученная модель способна четко разделять данные, упрощая выбор значения порога классификации без неприятных компромиссов

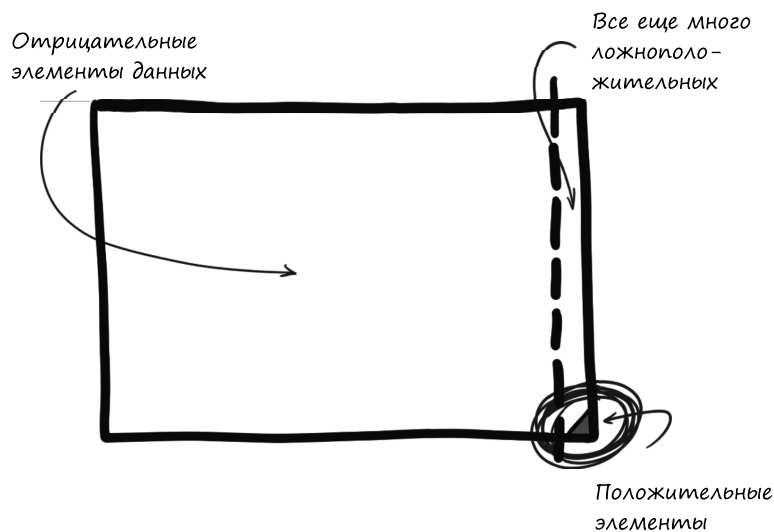


Рис. 12.16. Несбалансированный набор данных, примерно похожий на то, что мы имеем в наборе LUNA

Внесем ясность: когда мы сделаем то, что хотим, наша модель научится прекрасно справляться с таким дисбалансом данных. Может быть, нам бы и удалось

полностью обучить модель, не меняя балансировку, но это заняло бы миллиарды эпох¹. Вместо этого попробуем сделать обучающие данные более идеальными, изменив баланс классов, с помощью которых выполняется обучение.

12.4.1. Как сделать данные более «идеальными»

Лучше всего было бы увеличить долю положительных элементов данных. В начале обучения, когда мы переходим от рандомизированного хаоса к чему-то более организованному, наличие такого небольшого количества положительных обучающих образцов ведет к тому, что они будут незаметны.

Но вот методика того, как мы придем к успеху, довольно изощренная. Напомним: поскольку веса в сети изначально рандомизированы, выходные данные сети для каждой выборки также рандомизированы (но ограничены диапазоном $[0-1]$).

ПРИМЕЧАНИЕ

В качестве функции потерь будем использовать `nn.CrossEntropyLoss`, которая технически оперирует необработанными логитами, а не вероятностями класса. В рамках обсуждения мы проигнорируем это различие и предположим, что потери и дельта предсказания метки — это одно и то же.

Прогнозы, численно близкие к правильной метке, не приводят к значительному изменению весов сети, а прогнозы, значительно отличающиеся от правильного ответа, влияют на веса гораздо сильнее. Когда модель инициализируется со случайными весами, поэтому выходные данные тоже оказываются случайными и мы можем предположить, что из наших ~500 000 обучающих элементов (точнее, 495 958) образуются следующие приблизительные группы:

- 1) 250 000 отрицательных элементов будут классифицированы как отрицательные (от 0,0 до 0,5) и приведут к небольшому изменению весов сети в сторону предсказания отрицательного результата;
- 2) 250 000 отрицательных элементов будут классифицированы как положительные (от 0,5 до 1,0), что приведет к большому изменению весов сети, предсказывающих отрицательное значение;
- 3) 500 положительных выборок будут классифицированы как отрицательные, что приведет к изменению весов сети, предсказывающих положительное значение;
- 4) 500 положительных выборок будут классифицированы как положительные, и они почти не приведут к изменению весов сети.

¹ Неясно, правда ли это на самом деле, но выглядит правдоподобно.

ПРИМЕЧАНИЕ

Имейте в виду: прогнозы представляют собой действительные числа от 0,0 до 1,0 включительно, так что эти группы не будут иметь строгих границ.

И вот что самое интересное: группы 1 и 4 могут быть *любого размера* и они по-прежнему почти не будут влиять на обучение. Важно лишь то, что группы 2 и 3 должны достаточно противодействовать притяжению друг друга, чтобы предотвратить коллапс сети до вырожденного состояния «твердить одно и то же». Поскольку группа 2 в 500 раз больше, чем группа 3, и мы используем размер пакета, равный 32, то примерно $500 / 32 = 15$ пакетов должно пройти, чтобы встретился хотя бы один положительный элемент. Это означает, что 14 из 15 тренировочных пакетов будут на 100 % отрицательными и будут подтягивать все веса модели только к прогнозу отрицательного значения. Это одностороннее притяжение и вызывает дегенеративное поведение, которое мы наблюдаем.

Вместо этого нам хотелось бы иметь столько же положительных образцов, сколько и отрицательных. Таким образом, в первой части обучения половина меток будет классифицирована неправильно, а это означает, что группы 2 и 3 должны быть примерно равными по размеру. Мы также хотим быть уверены, что сеть получает пакеты как с отрицательными, так и с положительными элементами. Это позволит выровнять перетягивание каната, а смешение классов в партии даст модели хороший шанс научиться различать два класса. Поскольку в данных LUNA мало положительных элементов, нам придется размножить их и скормить модели много раз.

ДИСКРИМИНАЦИЯ

Под *дискриминацией* понимается способность различать два класса. Построение и обучение модели, которая умеет отличать настоящие узелки от нормальных анатомических структур — именно то, чем мы занимаемся в части II.

Есть и другие определения дискриминации. Для данного примера это неактуально, но существует более серьезная проблема с моделями, обученными на реальных данных. Если реальный набор данных собран из источников, имеющих дискриминационную предвзятость к реальному миру (скажем, расовая предвзятость в показателях арестов и осуждений или информация из социальных сетей), и она не исправлена во время подготовки набора данных или обучения, то итоговая модель будет воспроизводить те же ошибки, что и в обучающих данных. Модель тоже может научиться расизму.

Это означает, что почти любая модель, обученная на основе источников данных в интернете, будет так или иначе некорректна, если не подойти к удалению предубеждений из модели. Обратите внимание: как и наша цель в части II, это считается нерешенной проблемой.

Вспомните нашего профессора из главы 11, у которого на выпускном экзамене было 99 неверных ответов и один верный. В следующем семестре ему сказали: «Вам надо бы выровнять баланс верных и неверных ответов», и профессор решил в промежуточный семестр добавить тест с 99 верными ответами и одним неверным. И сказал, что проблема, дескать, решена.

Очевидно, правильный подход состоит в следующем: смешать верные и ложные ответы таким образом, чтобы учащиеся не могли для решения теста руководствоваться каким-то шаблоном. Но если учащийся улавливает шаблон вида «на нечетные вопросы пиши А, а на четные — Б», то система пакетной обработки, используемая PyTorch, не позволяет модели «замечать» или применять такой шаблон. Наш обучающий набор нужно будет отредактировать так, чтобы в нем чередовались положительные и отрицательные элементы, как показано на рис. 12.17.

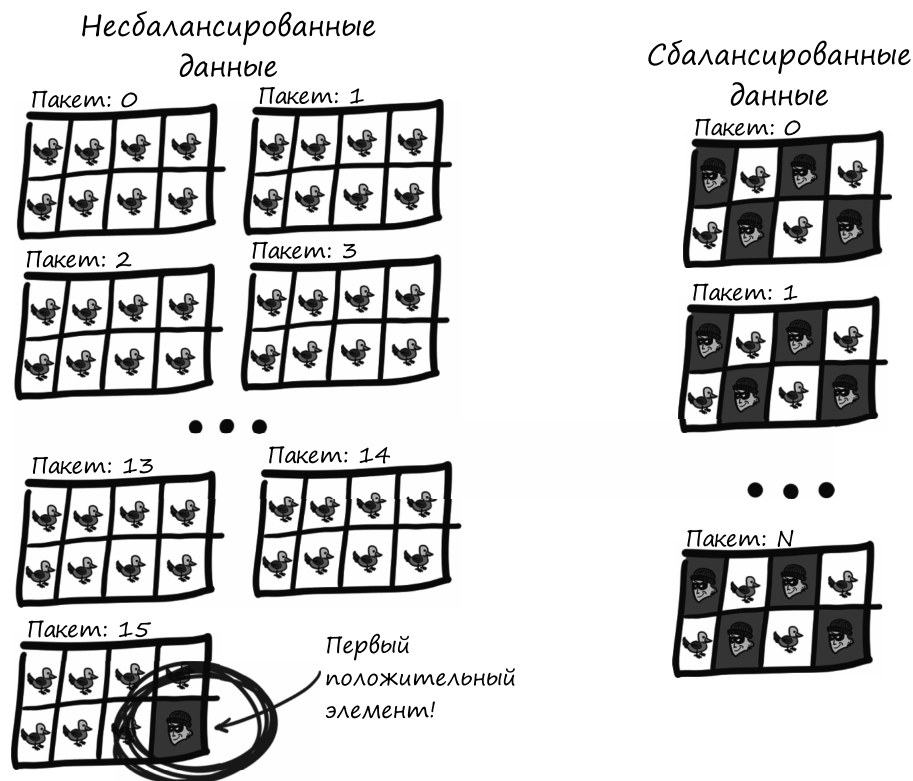


Рис. 12.17. Пакет за пакетом несбалансированных данных не будет иметь ничего, кроме отрицательных событий задолго до первого положительного события, в то время как сбалансированные данные могут чередоваться через каждую вторую выборку

Несбалансированные данные — это пресловутая иголка в стоге сена, о которой мы упоминали в начале главы 9. Если бы вам пришлось классифицировать все вручную, то вы, вероятно, прониклись бы тактикой Престона.

Мы не будем проводить балансировку проверочного набора. Наша модель должна хорошо работать в реальном мире, а он несбалансирован (в конце концов, именно из реального мира мы получили необработанные данные!).

Как достичь этого баланса? Обсудим варианты.

Сэмплеры могут изменять вид данных

Одним из необязательных аргументов класса `DataLoader` является `sampler=`. Он позволяет загрузчику данных переопределить порядок итераций набора данных, а также формировать, ограничивать или повторно выделять нужные данные по желанию. Это может быть невероятно полезно при работе с набором данных, который вы не можете контролировать. Взять общедоступный набор данных и изменить его под свои потребности гораздо проще, чем заново создавать этот набор данных с нуля.

Недостатком является то, что многие мутации, которые можно реализовать с помощью сэмплеров, требуют нарушения инкапсуляции базового набора данных. Например, предположим, у нас есть набор данных наподобие CIFAR-10 (www.cs.toronto.edu/~kriz/cifar.html), который состоит из десяти классов с одинаковым весом, и мы хотим, чтобы один класс (скажем, «самолет») теперь составлял 50 % всех обучающих изображений. Мы могли бы решить использовать `WeightedRandomSampler` (<http://mng.bz/8pIK>) и назначить индексам самолетов больший вес, но для этого нужно заранее знать, какие индексы являются самолетами.

Как мы уже говорили, API набора данных говорит лишь о том, что подклассы предоставляют методы `__len__` и `__getitem__`, но нет никакого способа узнать, какие элементы данных являются самолетами. Нам пришлось бы либо загружать все данные заранее с целью узнать класс элемента данных, либо нарушать инкапсуляцию и надеяться, что нужная нам информация будет легко получена из внутренней реализации подкласса набора данных.

Поскольку ни один из этих вариантов не слишком хорош в случаях, когда у нас есть прямой контроль над набором данных, код для части II реализует любое необходимое формирование данных внутри подклассов набора данных вместо того, чтобы полагаться на внешний сэмплер.

Реализация балансировки классов в наборе данных

Мы собираемся напрямую изменить наш `LunaDataset`, чтобы добиться сбалансированного соотношения положительных и отрицательных элементов.

Мы будем вести отдельные списки отрицательных и положительных обучающих элементов данных, а также чередовать их. Это предотвратит вырожденное поведение модели, которая дает хорошие оценки, просто отвечая `false` на каждый представленный образец. Кроме того, положительные и отрицательные классы будут перемешаны, так что обновления веса будут учитывать различия классов.

Добавим переменную `ratio_int` в `LunaDataset`, которая будет управлять меткой для N -го элемента, а также отслеживать их количество (листинг 12.6).

Листинг 12.6. `dsets.py:217`, класс `LunaDataset`

```
class LunaDataset(Dataset):
    def __init__(self,
                  val_stride=0,
                  isValSet_bool=None,
                  ratio_int=0,
                  ):
        self.ratio_int = ratio_int
        # ... строка 228
        self.negative_list = [
            nt for nt in self.candidateInfo_list if not nt.isNodule_bool
        ]
        self.pos_list = [
            nt for nt in self.candidateInfo_list if nt.isNodule_bool
        ]
        # ... строка 265

    def shuffleSamples(self):
        if self.ratio_int:
            random.shuffle(self.negative_list)
            random.shuffle(self.pos_list)
```

← Мы будем вызывать этот метод в начале каждой эпохи, чтобы рандомизировать порядок передачи элементов данных

Теперь у нас есть специальные списки для каждой метки. Используя их, становится намного проще возвращать метку, которую мы хотим для данного индекса, в набор данных. Чтобы убедиться в правильной индексации, нужно набросать желаемый порядок. Предположим, `ratio_int` равно 2, что означает соотношение отрицательных и положительных образцов 2:1. Это значило бы, что каждый третий элемент должен быть положительным:

Индекс DS	0	1	2	3	4	5	6	7	8	9	...
Этикетка	+	-	-	+	-	-	+	-	-	+	
Положительный индекс	0			1			2			3	
Отрицательный индекс		0	1		2	3		4	5		

Связь между индексом набора данных и положительным индексом проста: разделите индекс набора данных на 3, а затем округлите в меньшую сторону. С отрицательными индексами немного сложнее, поскольку нам нужно вычесть 1 из индекса набора данных, а затем также вычесть самый последний положительный индекс.

Реализация `LunaDataset` тогда выглядит следующим образом (листинг 12.7).

Листинг 12.7. `dsets.py:286, LunaDataset.__getitem__`

```
def __getitem__(self, ndx):
    if self.ratio_int:
        pos_ndx = ndx // (self.ratio_int + 1)

        if ndx % (self.ratio_int + 1):
            neg_ndx = ndx - 1 - pos_ndx
            neg_ndx %= len(self.negative_list)
            candidateInfo_tup = self.negative_list[neg_ndx]
        else:
            pos_ndx %= len(self.pos_list)
            candidateInfo_tup = self.pos_list[pos_ndx]
    else:
        candidateInfo_tup = self.candidateInfo_list[ndx]
```

Нулевое значение `ratio_int` означает, что мы не будем влиять на баланс

Ненулевой остаток означает, что это должен быть отрицательный элемент

Переполнение приводит к зацикливанию

Возвращает *N*-ю выборку, если классы не сбалансированы

Код может показаться немного странным, но если вы проверите его в деле, то он станет понятнее. Имейте в виду, что при низком соотношении положительные образцы закончатся раньше, чем переберется набор данных. Мы позаботимся об этом, взяв модуль `pos_ndx` перед индексацией в `self.pos_list`. С `neg_ndx` такого переполнения индекса никогда не должно происходить из-за большого количества отрицательных элементов, но мы все равно берем модуль на случай, если позже что-то изменится и переполнение вдруг появится.

Кроме того, мы изменим длину набора данных. Это не столь строго обязательно, но приятно ускорить выполнение эпох. Жестко закодируем параметр `__len__` равным 200 000 (листинг 12.8).

Листинг 12.8. `dsets.py:280, LunaDataset.__len__`

```
def __len__(self):
    if self.ratio_int:
        return 200000
    else:
        return len(self.candidateInfo_list)
```

Мы больше не привязаны к определенному количеству элементов данных, и вывод полной эпохи не имеет особого смысла, так как нам пришлось бы повторять положительные элементы множество раз, чтобы представить сбалансированный обучающий набор. Отбирая 200 000 элементов, мы сокращаем время между запуском обучающего прогона и получением результатов (а более быстрая обратная связь всегда приятна!), а количество элементов на эпоху получается вполне подходящим. Не стесняйтесь менять длину эпохи, если нужно.

Для полной ясности добавим также параметр командной строки (листинг 12.9).

Листинг 12.9. training.py:31, класс LunaTrainingApp

```
class LunaTrainingApp:
    def __init__(self, sys_argv=None):
        # ... строка 52
        parser.add_argument('--balanced',
                            help="Balance the training data to half positive, half negative.",
                            action='store_true',
                            default=False,
                            )
```

Затем мы передаем этот параметр в конструктор LunaDataset (листинг 12.10).

Листинг 12.10. training.py:137, LunaTrainingApp.initTrainDl

```
def initTrainDl(self):
    train_ds = LunaDataset(
        val_stride=10,
        isValSet_bool=False,
        ratio_int=int(self.cli_args.balanced),
    )
```

Здесь мы пользуемся тем, что значение True в Python приводится к 1

Все готово. Начнем!

12.4.2. Сравнение результатов обучения по сбалансированному и несбалансированному набору

Напоминаем, что наш несбалансированный тренировочный запуск дал такие результаты:

```
$ python -m p2ch12.training
...
E1 LunaTrainingApp
E1 trn      0.0185 loss,  99.7% correct, 0.0000 precision, 0.0000 recall,
  => nan f1 score
E1 trn_neg  0.0026 loss, 100.0% correct (494717 of 494743)
E1 trn_pos  6.5267 loss,   0.0% correct (0 of 1215)
...
E1 val      0.0173 loss,  99.8% correct, nan precision, 0.0000 recall,
  => nan f1 score
E1 val_neg  0.0026 loss, 100.0% correct (54971 of 54971)
E1 val_pos  5.9577 loss,   0.0% correct (0 of 136)
```

Но с аргументом --balanced мы видим следующее:

```
$ python -m p2ch12.training --balanced
...
E1 LunaTrainingApp
E1 trn      0.1734 loss,  92.8% correct, 0.9363 precision, 0.9194 recall,
  => 0.9277 f1 score
```

```

E1 trn_neg  0.1770 loss,  93.7% correct (93741 of 100000)
E1 trn_pos  0.1698 loss,  91.9% correct (91939 of 100000)
...
E1 val      0.0564 loss,  98.4% correct, 0.1102 precision, 0.7941 recall,
⇒ 0.1935 f1 score
E1 val_neg  0.0542 loss,  98.4% correct (54099 of 54971)
E1 val_pos  0.9549 loss,  79.4% correct (108 of 136)

```

Выглядит намного лучше! Мы потеряли около 5 % правильных ответов на отрицательных образцах, получив взамен 86 % правильных положительных ответов. Работает на твердую четверку!¹

Однако, как и в главе 11, этот результат обманчив. Поскольку отрицательных образцов в 400 раз больше, чем положительных, даже ошибка всего в 1 % означает, что мы неправильно классифицируем отрицательные образцы как положительные в четыре раза чаще, чем общее количество положительных образцов!

Тем не менее это все равно лучше, чем откровенно неправильное поведение из главы 11, и уж тем более лучше, чем случайное подбрасывание монеты. На самом деле наш код даже (почти) можно назвать полезным в реальных сценариях. Вспомним нашего переутомленного радиолога, изучающего каждую крупинку КТ. А теперь мы дали ему нечто, способное отсеять 95 % ложноположительных результатов. Это огромное подспорье, которое десятикратно увеличивает производительность человека.

Конечно, осталась еще серьезная проблема с 14 % пропущенных положительных образцов, с которой нам, вероятно, следует разобраться. Возможно, какие-то дополнительные эпохи обучения улучшат дело. Посмотрим (и снова рассчитывайте потратить не менее 10 минут на каждую эпоху):

```

$ python -m p2ch12.training --balanced --epochs 20
...
E2 LunaTrainingApp
E2 trn      0.0432 loss,  98.7% correct, 0.9866 precision, 0.9879 recall,
⇒ 0.9873 f1 score
E2 trn_ben  0.0545 loss,  98.7% correct (98663 of 100000)
E2 trn_mal  0.0318 loss,  98.8% correct (98790 of 100000)
E2 val      0.0603 loss,  98.5% correct, 0.1271 precision, 0.8456 recall,
⇒ 0.2209 f1 score
E2 val_ben  0.0584 loss,  98.6% correct (54181 of 54971)
E2 val_mal  0.8471 loss,  84.6% correct (115 of 136)
...
E5 trn      0.0578 loss,  98.3% correct, 0.9839 precision, 0.9823 recall,
⇒ 0.9831 f1 score
E5 trn_ben  0.0665 loss,  98.4% correct (98388 of 100000)

```

¹ И помните, что мы использовали набор из 200 000 элементов данных, а не более 500 000 элементов несбалансированного набора данных, поэтому получили результат более чем в два раза быстрее.


```

E5 trn_mal 0.0490 loss, 98.2% correct (98227 of 100000)
E5 val     0.0361 loss, 99.2% correct, 0.2129 precision, 0.8235 recall,
⇒ 0.3384 f1 score
E5 val_ben 0.0336 loss, 99.2% correct (54557 of 54971)
E5 val_mal 1.0515 loss, 82.4% correct (112 of 136)...
...
E10 trn     0.0212 loss, 99.5% correct, 0.9942 precision, 0.9953 recall,
⇒ 0.9948 f1 score
E10 trn_ben 0.0281 loss, 99.4% correct (99421 of 100000)
E10 trn_mal 0.0142 loss, 99.5% correct (99530 of 100000)
E10 val     0.0457 loss, 99.3% correct, 0.2171 precision, 0.7647 recall,
⇒ 0.3382 f1 score
E10 val_ben 0.0407 loss, 99.3% correct (54596 of 54971)
E10 val_mal 2.0594 loss, 76.5% correct (104 of 136)
...
E20 trn     0.0132 loss, 99.7% correct, 0.9964 precision, 0.9974 recall,
⇒ 0.9969 f1 score
E20 trn_ben 0.0186 loss, 99.6% correct (99642 of 100000)
E20 trn_mal 0.0079 loss, 99.7% correct (99736 of 100000)
E20 val     0.0200 loss, 99.7% correct, 0.4780 precision, 0.7206 recall,
⇒ 0.5748 f1 score
E20 val_ben 0.0133 loss, 99.8% correct (54864 of 54971)
E20 val_mal 2.7101 loss, 72.1% correct (98 of 136)

```

Да уж. Придется прокрутить много текста, чтобы добраться до интересующих нас чисел. Пойдем глубже и рассмотрим числа `val_mal XX.X% correct` (или сразу перейдем к графику `TensorBoard` в следующем подразделе). После эпохи 2 мы были на уровне 87,5 %, в эпоху 5 достигли пика 92,6%, а затем к 20-й эпохе упали до 86,8 % — хуже эпохи 2!

ПРИМЕЧАНИЕ

Как упоминалось ранее, стоит ожидать, что каждый запуск будет вести себя по-своему из-за случайной инициализации весов сети и случайного выбора и упорядочения обучающих выборок для каждой эпохи.

Не похоже, чтобы в тренировочном наборе была та же проблема. Отрицательные обучающие выборки правильно классифицируются в 98,8 % случаев, а положительные — в 99,1 %. Что происходит?

12.4.3. Распознавание симптомов переобучения

Налицо явные признаки *переобучения*. Посмотрим на график потерь на положительных образцах на рис. 12.18.

Мы видим, что потери при обучении для наших положительных элементов почти равны нулю, то есть каждый положительный элемент в обучающем наборе получает почти идеальный прогноз. Однако наши потери при валидации

для положительных образцов увеличиваются, а это значит, что реальная производительность, скорее всего, ухудшается. В этот момент часто бывает лучше остановить сценарий обучения, так как модель больше не улучшается.

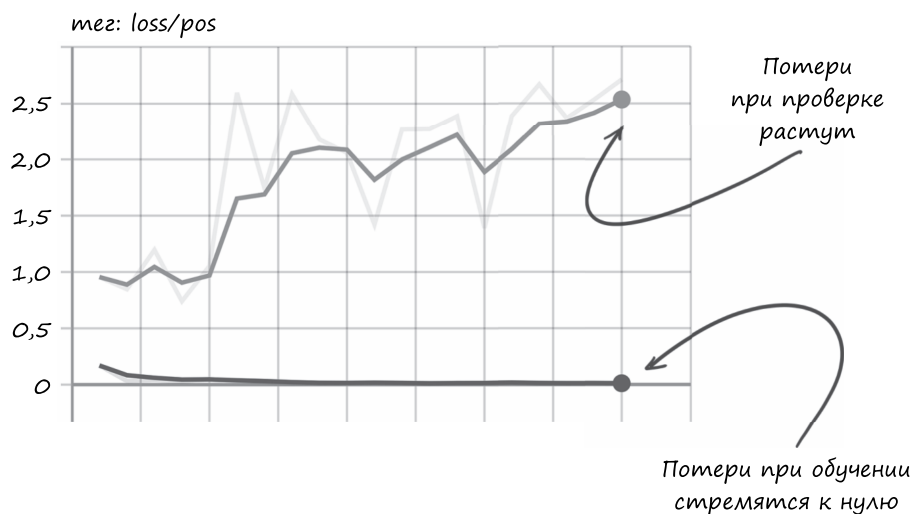


Рис. 12.18. Положительные потери демонстрируют явные признаки переобучения, поскольку потери при обучении и при проверке движутся в разные стороны

СОВЕТ

Как правило, если производительность вашей модели улучшается на обучающем наборе, а на проверочном наборе ухудшается — значит, модель начала переобучаться.

Теперь важно рассмотреть правильные метрики, поскольку эта тенденция проявляется только в потерях на *положительных* данных. Если мы посмотрим на общие потери, все выглядит прекрасно! Это связано с тем, что наш проверочный набор не сбалансирован, вследствие чего в общих потерях преобладают отрицательные образцы. Как показано на рис. 12.19, для отрицательных образцов мы не наблюдаем такого же расходящегося поведения и потери выглядят великолепно! Дело в том, что у нас в 400 раз больше отрицательных образцов, поэтому модели гораздо труднее запомнить отдельные детали. Однако положительный обучающий набор содержит всего 1215 элементов. Мы передаем их модели по несколько раз, однако это не усложняет их запоминание. Модель переходит от обобщенных принципов к запоминанию особенностей тех 1215 образцов и говорит, что все непохожее на какой-то из тех немногих элементов является отрицательным. Сюда входят как отрицательные обучающие элементы, так и все данные в проверочном наборе (как положительные, так и отрицательные).

В то же время некоторое обобщение модель все же делает, поскольку мы правильно классифицируем около 70 % положительных элементов в проверочном наборе. Нам просто нужно изменить способ обучения модели, чтобы обучающий и проверочный наборы создавали тренд в правильном направлении.

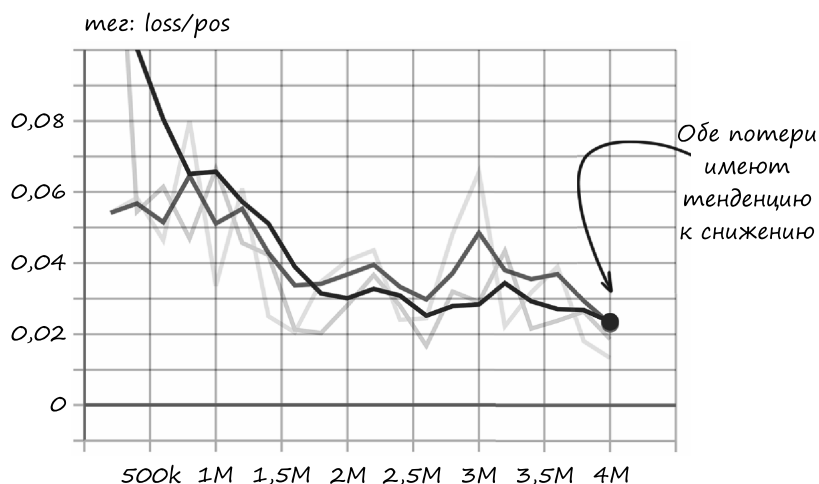


Рис. 12.19. Отрицательные потери не показывают признаков переобучения

12.5. ВЕРНЕМСЯ К ПРОБЛЕМЕ ПЕРЕОБУЧЕНИЯ

Мы уже касались понятия переобучения в главе 5, а теперь пришло время более подробно рассмотреть, как действовать в этой довольно распространенной ситуации. Наша цель при обучении модели — научить ее распознавать *общие свойства* интересующих нас классов, выраженные в нашем наборе данных. Эти свойства присутствуют в некоторых или во всех элементах данных класса и могут быть *обобщены* и использованы для прогнозирования элементов, на которых не проводилось обучение. Когда модель начинает изучать *определенные свойства* обучающей выборки, происходит переобучение и модель начинает терять способность к обобщению. Если это объяснение слишком абстрактно, то прибегнем к другой аналогии.

12.5.1. Модель прогнозирования с переобучением по возрасту

Представим, что у нас есть модель, которая принимает в качестве входных данных изображение человеческого лица и выводит возраст человека в годах. Хорошая модель улавливает признаки возраста, такие как морщины, седые

волосы, прическа, выбор одежды и т. п., и использует их для построения общей картины того, как выглядит человек в разном возрасте. Получив новую фотографию, модель должна увидеть такие вещи, как «консервативная стрижка», «очки для чтения» и «морщины», и сделать вывод, что человеку «около 65 лет».

Переобученная модель, напротив, запоминает конкретных людей и их особые признаки. «Эта стрижка и эти очки означают, что это Фрэнк. Ему 62,8 года». «О, этот шрам означает, что это Гарри. Ему 39,3 года» и т. д. Увидев нового человека, модель не узнает его и совершенно не будет знать, какой возраст предсказать.

Хуже того, если показать фотографию Фрэнка-младшего (который как две капли похож на отца, по крайней мере в очках!), то модель скажет: «Я думаю, это Фрэнк. Ему 62,8 года». И неважно, что сын моложе на 25 лет!

Переобучение обычно происходит из-за слишком малого количества обучающих элементов по сравнению со способностью модели просто запоминать ответы. Среднестатистический человек может запомнить дни рождения своих ближайших родственников, но ему придется прибегнуть к обобщениям при предсказании возраста группы людей размером с небольшую деревню.

Наша модель «лицо — возраст» способна просто запоминать фотографии всех, кто не выглядит точно на свой возраст. Как мы обсуждали в части I, мощность модели — несколько абстрактное понятие, но примерно зависящее от количества параметров модели и от эффективности их использования. Когда мощность модели велика по сравнению с объемом данных, необходимых для запоминания сложных выборок из обучающего набора, вполне вероятно, что модель начнет переобучаться на этих более сложных наборах данных.

12.6. ПРЕДОТВРАЩЕНИЕ ПЕРЕОБУЧЕНИЯ ПУТЕМ УВЕЛИЧЕНИЯ НАБОРА ДАННЫХ

Пришло время обучить не хорошую, а отличную модель. Нам осталось пройти последний шаг на рис. 12.20.

Мы *дополняем* набор данных, применяя синтетические изменения к отдельным элементам, в результате чего получается новый набор данных, превышающий по размеру исходный. Обычно цель дополнения — получить синтетический набор, который остается репрезентативным для того же общего класса, что и исходный, но который нельзя тривиально запомнить вместе с оригиналом. При правильном выполнении это увеличение может увеличить тренировочный набор настолько, что модель не сможет его запомнить, и тогда она будет вынуждена больше полагаться на обобщение, а это именно то, что нам нужно. Это особенно полезно при работе с ограниченными данными, как мы видели в подразделе 12.4.1.

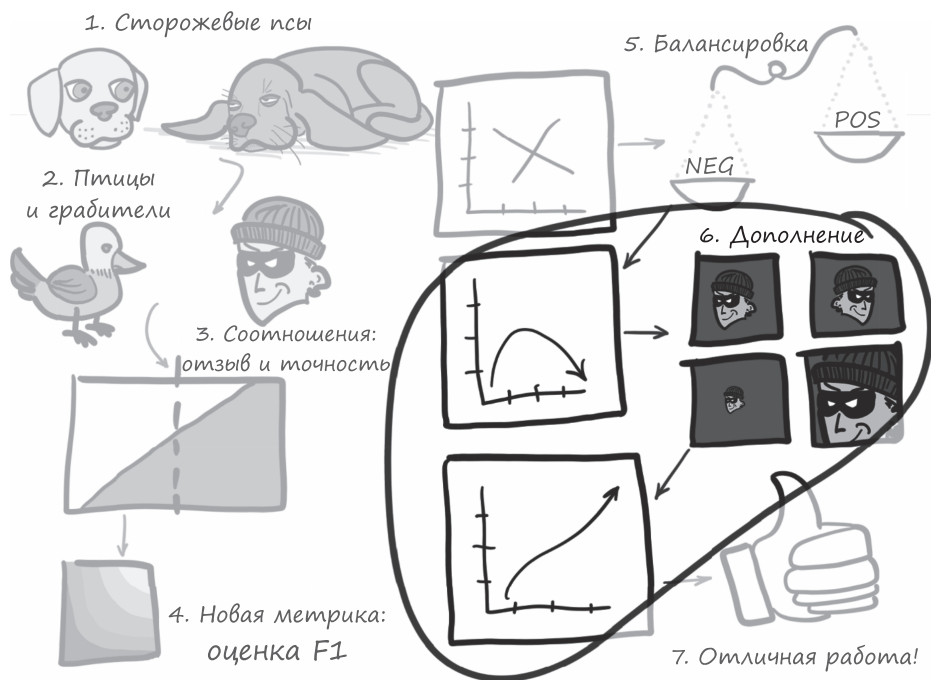


Рис. 12.20. Задачи этой главы. Выделен этап дополнения данных

Конечно, не все дополнения одинаково полезны. В примере с моделью предсказания возраста мы могли бы попросту изменить красный канал четырех угловых пикселей каждого изображения на случайное значение от 0 до 255, что привело бы к получению набора данных в 4 миллиарда раз больше исходного. Конечно, полезного в этом мало, поскольку модель может достаточно быстро научиться игнорировать красные точки в углах изображения, а остальную часть изображения так же легко запомнить, как и исходное изображение без дополнений. Другой подход — отражение изображения слева направо. Это позволяет получить набор данных вдвое больше исходного, но каждое изображение будет намного полезнее для учебных целей. Общие свойства старения не коррелируют слева направо, поэтому зеркальное изображение остается репрезентативным. Кроме того, лица редко бывают идеально симметричными, вследствие чего отраженное лицо не будет принято за оригинал.

12.6.1. Методы дополнения данных

Мы собираемся реализовать пять типов дополнения данных. Наша реализация позволит нам экспериментировать с любым из них или со всеми по отдельности или в совокупности. Опишем эти методы:

- зеркальное отражение изображения вверх-вниз, влево-вправо и/или вперед-назад;
- сдвиг изображения на несколько вокселей;
- масштабирование изображения вверх или вниз;
- вращение изображения вокруг оси «голова — нога»;
- добавление шума.

Для каждого метода нужно убедиться, что выбранный подход сохраняет репрезентативный характер измененного обучающего элемента, но в то же время отличается достаточно, чтобы этот элемент был полезен для обучения.

Мы определим функцию `getCtAugmentedCandidate`, которая отвечает за получение нашего стандартного фрагмента КТ с кандидатом и его изменение. В качестве основного подхода мы определим матрицу аффинного преобразования (<http://mng.bz/Edxq>) и используем ее с PyTorch `affine_grid` (<https://pytorch.org/docs/stable/nn.html#affine-grid>) и `grid_sample` (https://pytorch.org/docs/stable/nn.html#torch.nn.functional.grid_sample) для повторной выборки нашего кандидата (листинг 12.11).

Листинг 12.11. `dsets.py:149, def getCtAugmentedCandidate`

```
def getCtAugmentedCandidate(
    augmentation_dict,
    series_uid, center_xyz, width_irc,
    use_cache=True):
    if use_cache:
        ct_chunk, center_irc = \
            getCtRawCandidate(series_uid, center_xyz, width_irc)
    else:
        ct = getCt(series_uid)
        ct_chunk, center_irc = ct.getRawCandidate(center_xyz, width_irc)

    ct_t = torch.tensor(ct_chunk).unsqueeze(0).unsqueeze(0).to(torch.float32)
```

Сначала мы получаем `ct_chunk` либо из кэша, либо напрямую, загрузив КТ (это пригодится, когда мы создадим собственные центры-кандидаты), а затем преобразуем его в тензор. Далее идет аффинная сетка и код выборки (листинг 12.12).

Листинг 12.12. `dsets.py:162, def getCtAugmentedCandidate`

```
transform_t = torch.eye(4)
# ...      ←      Изменения в transform_tensor отразятся здесь
# ... строка 195
affine_t = F.affine_grid(
    transform_t[:3].unsqueeze(0).to(torch.float32),
    ct_t.size(),
    align_corners=False,
)
```

```
augmented_chunk = F.grid_sample(
    ct_t,
    affine_t,
    padding_mode='border',
    align_corners=False,
).to('cpu')
# ... строка 214
return augmented_chunk[0], center_irc
```

Без каких-либо дополнений эта функция мало что даст. Посмотрим, что нужно для добавления преобразований.

ПРИМЕЧАНИЕ

Важно структурировать конвейер данных таким образом, чтобы этапы кэширования выполнялись до дополнения данных! В противном случае ваши данные дополнятся, а затем будут сохранены в таком состоянии, а это противоречит цели.

Отражение

При отражении элемента мы сохраняем значения пикселей точно такими же, меняя только ориентацию изображения. Поскольку нет сильной корреляции между ростом опухоли и «влево-вправо» или «вперед-назад», мы должны иметь возможность переворачивать их, не изменяя репрезентативного характера выборки. Однако ось индекса (в координатах пациента это ось Z) соответствует направлению силы тяжести у человека в вертикальном положении, так что существует вероятность разницы в верхней и нижней частях опухоли. Мы предположим, что это нормально, поскольку быстрое визуальное исследование не показывает каких-либо грубых отклонений. Если бы мы работали над клинически значимым проектом, то нам нужно было бы подтвердить это предположение у эксперта (листинг 12.13).

Листинг 12.13. dsets.py:165, def getCtAugmentedCandidate

```
for i in range(3):
    if 'flip' in augmentation_dict:
        if random.random() > 0.5:
            transform_t[i,i] *= -1
```

Функция `grid_sample` масштабирует диапазон $[-1, 1]$ на размер как старого, так и нового тензоров (масштабирование происходит неявно, если размеры различаются). Это масштабирование диапазона означает, что для отражения нам достаточно умножить соответствующий элемент матрицы преобразования на -1 .

Сдвиг на случайную величину

Перемещение узелка-кандидата не должно иметь большого значения, поскольку его форма при этом не меняется, зато модель окажется более устойчивой к несовершенным центрированным узелкам. Более существенное значение имеет

тот факт, что смещение может быть не целым числом вокселей. Вместо этого данные передискретизируются с помощью трилинейной интерполяции, что может привести к небольшому размытию. Воксели на краю фрагмента будут повторяться, и визуально это выглядит как смазанный участок вдоль границы (листинг 12.14).

Листинг 12.14. `dssets.py:165, def getCtAugmentedCandidate`

```
for i in range(3):
    # ... строка 170
    if 'offset' in augmentation_dict:
        offset_float = augmentation_dict['offset']
        random_float = (random.random() * 2 - 1)
        transform_t[i,3] = offset_float * random_float
```

Обратите внимание: наш параметр 'offset' представляет собой максимальное смещение, выраженное в той же шкале, что и диапазон $[-1, 1]$, который ожидает функция выборки сетки.

Масштабирование

Небольшое масштабирование изображения очень похоже на зеркальное отражение и смещение. Оно тоже может породить повторяющиеся краевые воксели, которые мы только что упомянули (листинг 12.15).

Листинг 12.15. `dssets.py:165, def getCtAugmentedCandidate`

```
for i in range(3):
    # ... строка 175
    if 'scale' in augmentation_dict:
        scale_float = augmentation_dict['scale']
        random_float = (random.random() * 2 - 1)
        transform_t[i,i] *= 1.0 + scale_float * random_float
```

Поскольку `random_float` преобразуется в диапазон $[-1, 1]$, на самом деле не имеет значения, прибавляем ли мы `scale_float * random_float` к 1.0 или вычитаем его из него.

Вращение

Вращение — первый метод дополнения, который мы будем использовать, но здесь нужно тщательно изучить данные и убедиться, что преобразование не нарушает репрезентативность элемента. Напомним, что наши срезы КТ имеют одинаковые интервалы вдоль строк и столбцов (оси X и Y), но в индексном (или Z) направлении воксели некубические. То есть мы не можем рассматривать эти оси как взаимозаменяемые.

Один из вариантов — передискретизировать наши данные, чтобы разрешение по оси индекса стало таким же, как и по двум другим, но это неверное решение,

поскольку данные по этой оси будут очень размытыми. Даже если мы будем интерполировать больше вокселей, достоверность данных останется низкой. Вместо этого мы будем рассматривать эту ось как особый случай и ограничим вращение плоскостью XY (листинг 12.16).

Листинг 12.16. `dssets.py:181, def getCtAugmentedCandidate`

```
if 'rotate' in augmentation_dict:
    angle_rad = random.random() * math.pi * 2
    s = math.sin(angle_rad)
    c = math.cos(angle_rad)

    rotation_t = torch.tensor([
        [c, -s, 0, 0],
        [s, c, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1],
    ])

    transform_t @= rotation_t
```

Шум

Наша последняя техника дополнения и другие разнятся тем, что она буквально портит данные, в отличие от отражения или поворота. Если мы добавим в элемент слишком много шума, то он заглушит реальные данные и сделает классификацию невозможной. Хотя смещение и масштабирование элемента тоже делает нечто подобное, если переборщить с преобразованием значения, но мы выбрали значение, которое негативно скажется только на крайней части элемента. А шум влияет на все изображение (листинг 12.17).

Листинг 12.17. `dssets.py:208, def getCtAugmentedCandidate`

```
if 'noise' in augmentation_dict:
    noise_t = torch.randn_like(augmented_chunk)
    noise_t *= augmentation_dict['noise']

    augmented_chunk += noise_t
```

Другие типы дополнений просто увеличивают размер набора данных. А шум *усложняет* модели работу. Мы вернемся к этому, как только увидим результаты обучения.

Изучение дополнительных кандидатов

Результат наших усилий показан на рис. 12.21. На верхнем левом изображении показан положительный кандидат без дополнений, а следующие пять картинок показывают влияние каждого типа дополнения по отдельности. Наконец, в нижней строке трижды показан объединенный результат.

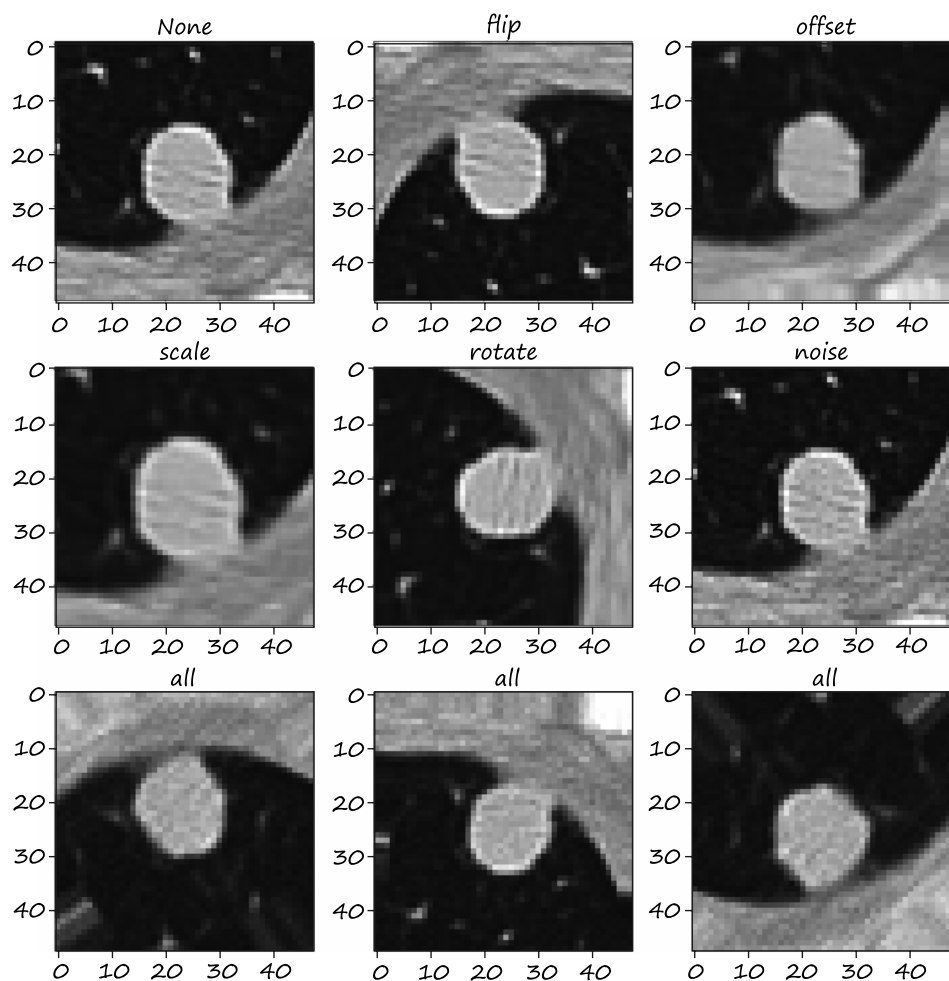


Рис. 12.21. Различные типы дополнения, выполненные на положительном образце узелка

Поскольку при каждом вызове `__getitem__` на дополненном наборе данных изменения применяются случайным образом, каждое изображение в нижней строке выглядит по-своему. Это означает еще и то, что почти невозможно снова сгенерировать точно такое же изображение!

Кроме того, важно помнить, что иногда аугментация `'flip'`, по сути, на самом деле *не выполняет* отражение. Всегда возвращать перевернутые изображения — это то же самое, что никогда не переворачивать. Теперь посмотрим, что из описанного позволит улучшить результаты.

12.6.2. Наблюдение за улучшением данных после дополнения

Попробуем обучить дополнительные модели, по одной для каждого типа дополнения, о которых мы поговорили в предыдущем подразделе, а также прогоним модель, объединяющую сразу все виды. Как только обучение закончится, посмотрим на числа в TensorBoard.

Чтобы иметь возможность включать и выключать определенные типы дополнений, нам нужно предоставить конструкцию `augmentation_dict` нашему интерфейсу командной строки. Аргументы будут добавляться к программе с помощью вызовов `parser.add_argument` (они не показаны, но аналогичны уже имеющимся в нашей программе). Аргументы затем передаются в код, который фактически создает `augmentation_dict` (листинг 12.18).

Листинг 12.18. `training.py:105, LunaTrainingApp.__init__`

```
self.augmentation_dict = {}
if self.cli_args.augmented or self.cli_args.augment_flip:
    self.augmentation_dict['flip'] = True
if self.cli_args.augmented or self.cli_args.augment_offset:
    self.augmentation_dict['offset'] = 0.1
if self.cli_args.augmented or self.cli_args.augment_scale:
    self.augmentation_dict['scale'] = 0.2
if self.cli_args.augmented or self.cli_args.augment_rotate:
    self.augmentation_dict['rotate'] = True
if self.cli_args.augmented or self.cli_args.augment_noise:
    self.augmentation_dict['noise'] = 25.0
```

Эти значения были выбраны эмпирическим путем, чтобы иметь разумное влияние, но, вероятно, существуют и лучшие значения

Теперь, когда у нас есть готовые аргументы командной строки, вы можете либо запустить следующие команды, либо вернуться к `p2_run_everything.ipynb` и запустить ячейки с 8-й по 16-ю. В любом случае выполнение этих команд займет значительное время:

```
$ .venv/bin/python -m p2ch12.prepcache
$ .venv/bin/python -m p2ch12.training --epochs 20 \
  --balanced sanity-bal
$ .venv/bin/python -m p2ch12.training --epochs 10 \
  --balanced --augment-flip sanity-bal-flip
$ .venv/bin/python -m p2ch12.training --epochs 10 \
  --balanced --augment-shift sanity-bal-shift
$ .venv/bin/python -m p2ch12.training --epochs 10 \
  --balanced --augment-scale sanity-bal-scale
```

Кэш достаточно подготовить лишь раз для каждой главы

Возможно, вы выполняли этот запуск ранее в этой главе. В этом случае нет необходимости перезапускать его

```
$ .venv/bin/python -m p2ch12.training --epochs 10 \
    --balanced --augment-rotate sanity-bal-rotate

$ .venv/bin/python -m p2ch12.training --epochs 10 \
    --balanced --augment-noise sanity-bal-noise

$ .venv/bin/python -m p2ch12.training --epochs 20 \
    --balanced --augmented sanity-bal-aug
```

Пока модель работает, мы можем запустить TensorBoard. Настроим все так, чтобы отображались только последние запуски, изменив параметр `logdir` следующим образом: `../path/to/tensorboard --logdir runs/p2ch12`.

В зависимости от вашего оборудования обучение может занять больше или меньше времени. Вы можете пропустить задания по переворачиванию, смещению и масштабированию, а также сократить первый и последний прогоны до 11 эпох, если нужно ускорить процесс. Мы выбрали 20 прогонов для различимости, но 11 тоже должно хватить.

Если вы доведете все до конца, то в TensorBoard отобразятся данные, показанные на рис. 12.22. Мы уберем отображение всего, кроме данных проверки, чтобы на графиках не было мешанины. Просматривая данные в реальном времени, вы также можете изменить значение сглаживания, что может помочь уточнить линии тренда. Взгляните на рисунок, а затем мы разберем его более подробно.

Первое, на что следует обратить внимание на верхнем левом графике (`tag: correct/all`), — это то, что отдельные типы дополнений работают совершенно беспорядочно. Прогоны без дополнений или со всеми дополнениями находятся на противоположных сторонах этого беспорядка. Получается, дополнения, будучи вместе, дают лучший результат, чем когда они проходят по отдельности. Интересно еще и то, что прогон со всеми дополнениями дает гораздо больше неправильных ответов. Хотя в целом это плохо, но если мы посмотрим на правый столбец изображений (данные по положительным кандидатам, которые нам действительно интересны, так как являются узелками), то увидим, что наша полностью дополненная модель намного лучше находит положительных кандидатов. Отклик у полностью дополненной модели отличный! Вдобавок она гораздо лучше работает с точки зрения переобучения. А модель без дополнений со временем ухудшается.

Следует отметить одну интересную вещь: модель с шумами хуже идентифицирует узелки, чем модель без дополнений. Это имеет смысл, ведь мы говорили, что шум усложняет работу модели.

Еще одна интересная вещь, которую можно увидеть в реальных данных (хотя тут она несколько теряется в беспорядке), заключается в том, что модель с вращением почти так же хороша, как модель со всеми дополнениями, с точки зрения точности и отклика. Поскольку наша метрика F1 имеет ограниченную точность

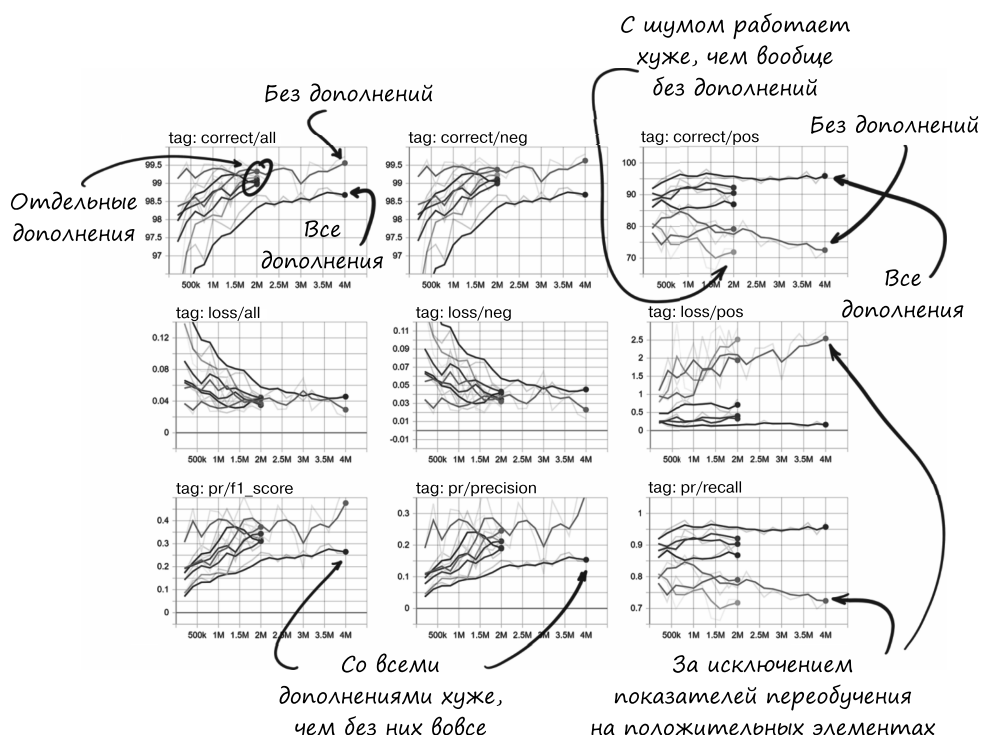


Рис. 12.22. Процент правильно классифицированных элементов, потери, метрика F1, точность и отклик для проверочного набора из сетей, обученных с помощью различных схем дополнения

(из-за большего количества отрицательных элементов), модель с вращением также имеет лучшую метрику F1.

В будущем мы будем работать с моделью со всеми дополнениями, поскольку наша задача требует высокого отклика. Метрика F1 по-прежнему будет использоваться для определения лучшей эпохи из представленных. В реальном проекте мы могли бы посвятить дополнительное время изучению различных комбинаций дополнений и значений их параметров, чтобы получить еще более хорошие результаты.

12.7. ИТОГИ ГЛАВЫ

В этой главе мы потратили немало времени и сил на то, чтобы переформулировать представление о производительности нашей модели. Плохие методы оценки могут легко ввести вас в заблуждение, и крайне важно иметь

интуитивное понимание факторов, позволяющих определить правильную оценку. Когда эти основы усвоены, становится намного легче заметить нечто, идущее не по плану.

Мы также узнали, как обращаться с источниками данных, если данных в них недостаточно. Возможность синтезировать репрезентативные обучающие данные невероятно полезна. Ситуации, когда у нас слишком много обучающих данных, на самом деле довольно редки!

Теперь, когда мы получили хорошо работающий классификатор, можно начать автоматический поиск узлов-кандидатов для классификации. Этим и займемся в главе 13. Затем в главе 14 вернем этих кандидатов обратно в разработанный классификатор и рискнем создать еще один классификатор, который должен будет отличать злокачественные узелки от доброкачественных.

12.8. УПРАЖНЕНИЯ

1. Метрику F1 можно обобщить для поддержки значений, отличных от 1.
 - А. Прочитайте статью https://en.wikipedia.org/wiki/F1_score и внедрите метрики F2 и F0.5.
 - Б. Определите, какая из метрик — F1, F2 или F0.5 — лучше подходит для данного проекта. Отследите это значение, сравните и сопоставьте его с метрикой F1.¹
2. Реализуйте подход `WeightedRandomSampler` для балансировки положительных и отрицательных обучающих элементов для `LunaDataset` с параметром `ratio_int`, равным 0.
 - А. Как вы получили необходимую информацию о классе каждого образца?
 - Б. Какой подход был проще? Какой дал более читаемый код?
3. Поэкспериментируйте с различными схемами балансировки классов.
 - А. Какое соотношение приводит к наилучшему результату после двух эпох? После 20?
 - Б. Что, если отношение является функцией `epoch_ndx`?
4. Поэкспериментируйте с различными подходами к дополнению данных.
 - А. Можно ли сделать какой-либо из существующих подходов более агрессивным (шум, смещение и т. д.)?

¹ И да, это намек, что F1 здесь не фаворит!

- Б. Как на результаты тренировок влияет включение шумоподавления?
 - Существуют ли другие значения, которые изменяют этот результат?
- В. Исследуйте методы дополнения данных, которые использовались в других проектах. Применимы ли они здесь?
 - Внедрите аугментацию «смешивания» для положительных кандидатов. Стало ли лучше?
- 5. Измените начальную нормализацию с `nn.BatchNorm` на что-то свое и повторно обучите модель.
 - А. Можно ли улучшить результаты с помощью фиксированной нормализации?
 - Б. Какое смещение и масштаб нормализации имеет смысл использовать?
 - В. Помогают ли нелинейные нормализации, такие как квадратные корни?
- 6. Какие еще типы данных может отображать `TensorBoard`, кроме тех, которые мы рассмотрели здесь?
 - А. Можете ли вы отображать информацию о весе вашей сети?
 - Б. Что можно сказать о промежуточных результатах запуска вашей модели на конкретном образце?
 - Как влияет на работу модель ее оборачивание в экземпляр `nn.Sequential`?

12.9. РЕЗЮМЕ

- Бинарные метки и пороговое значение бинарной классификации разделили исходный набор данных на четыре квадранта: истинно положительные, истинно отрицательные, ложноотрицательные и ложноположительные. Эти четыре параметра лежат в основе метрик эффективности, которые мы ввели.
- Отклик — способность модели максимизировать истинно положительные результаты. Положительная классификация каждого элемента гарантирует идеальный отклик, поскольку все правильные ответы будут отмечены, но при этом снижается точность.
- Точность — способность модели минимизировать ложные срабатывания. Если не выбрать ничего, то это гарантирует идеальную точность, поскольку неправильных ответов не было, но также свидетельствует о плохом отклике.
- Метрика F1 объединяет точность и полноту в единую метрику, которая описывает производительность модели. Мы используем метрику F1, чтобы определить, какое влияние изменения в алгоритме обучения или модели оказывают на производительность.

- Балансировка обучающего набора до равного количества положительных и отрицательных элементов во время обучения может привести к улучшению работы модели (определяемой как наличие положительной, увеличивающейся метрики F1).
- Дополнение данных — это взятие имеющихся реальных данных и такое их изменение, чтобы полученный дополненный элемент нетривиально отличался от исходного, но оставался репрезентативным для элементов того же класса. Это позволяет избежать переобучения в случаях, когда данные ограничены.
- К общим стратегиям дополнения данных относят изменение ориентации, зеркальное отражение, изменение масштаба, смещение и добавление шума. В зависимости от особенностей проекта, могут использоваться и другие, более конкретные стратегии.

13

Поиск потенциальных узелков с помощью сегментации

В этой главе

- ✓ Сегментация данных с использованием попиксельной модели.
- ✓ Сегментация с помощью U-Net.
- ✓ Концепция прогноза по маске с использованием функции Dice loss.
- ✓ Оценка производительности модели сегментации.

В предыдущих четырех главах мы многого достигли. Мы узнали о компьютерной томографии и опухолях в легких, наборах данных и загрузчиках данных, метриках и мониторинге. Мы также *применили* многое из того, что узнали в части I, и у нас получился работающий классификатор. Но мы все еще работаем в несколько искусственной среде, поскольку классификатору требуется предварительно аннотированная информация об узелках-кандидатах. У нас нет хорошего способа получить аннотации автоматически. Если мы передадим всю КТ в модель, а это перекрывающиеся фрагменты данных размером $32 \times 32 \times 32$, то получим $31 \times 31 \times 7 = 6727$ фрагментов на КТ, то есть примерно в десять раз больше числа аннотированных элементов, которые у нас есть. Нам нужно перекрыть края, так как классификатор ожидает, что кандидат в узелки будет находиться в центре, и непоследовательное позиционирование, вероятно, приведет к проблемам.

Как мы обсуждали в главе 9, в проекте должно быть несколько этапов, которые вместе должны решить задачу локализации возможных узелков, а также

определения их злокачественности. У практиков принято делить работу на стадии, но в исследованиях глубокого обучения чаще пытаются добиться от отдельных моделей возможности решать сложные задачи сквозным образом. Многоступенчатый дизайн проекта, который мы используем в этой книге, дает нам хороший повод этап за этапом вводить новые концепции.

13.1. ДОБАВИМ В ПРОЕКТ ВТОРУЮ МОДЕЛЬ

В предыдущих двух главах мы работали над этапом 4 нашего плана, показанного на рис. 13.1, — классификацией. В этой главе мы вернемся не на один, а на два шага назад. Нам нужно найти способ сообщить нашему классификатору, куда вообще смотреть. Для этого возьмем необработанные компьютерные томограммы и определим на них все, что может быть узелком¹. На рис. 13.1 это выделенный этап 2. Чтобы найти возможные узелки, мы должны пометить воксели, которые выглядят так, будто могут быть частью узелка. Этот процесс называется сегментацией.

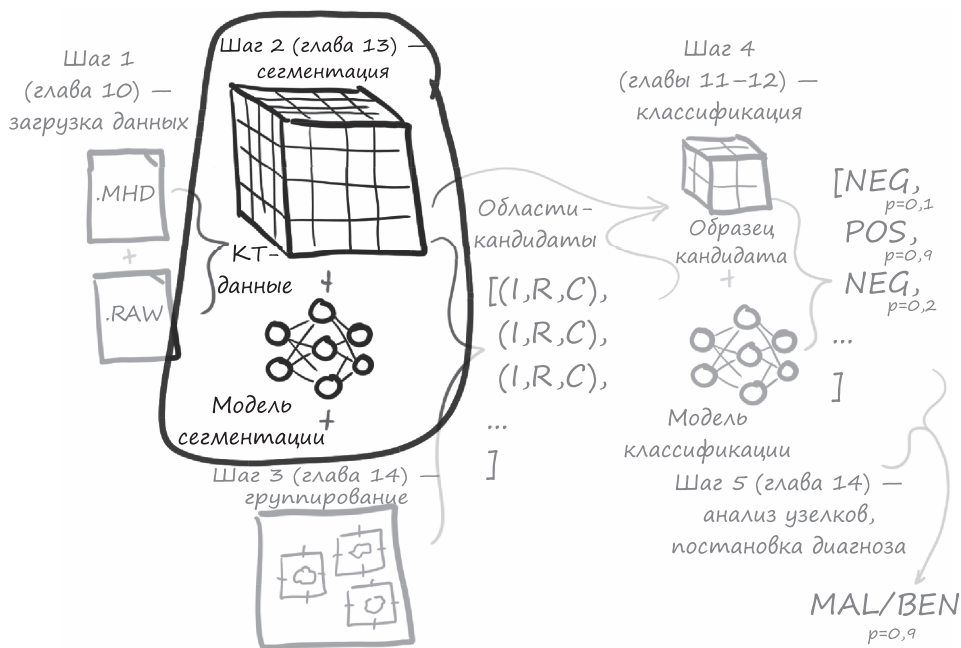


Рис. 13.1. Наш комплексный проект по обнаружению рака легких. Выделена основная тема данной главы: этап 2 — сегментация

¹ Наверняка мы отметим довольно много вещей, которые не являются узелками, поэтому нужно будет использовать этап классификации, чтобы уменьшить их количество.

Затем в главе 14 мы рассмотрим этап 3 и создадим мост, преобразовав маски сегментации с изображения в аннотации местоположения.

К концу этой главы мы создадим новую модель с архитектурой, которая сможет выполнять попиксельную маркировку или сегментацию.

Код, с помощью которого мы это реализуем, будет очень похож на код из предыдущей главы, особенно если мы сосредоточимся на более крупной структуре. Все изменения, которые мы собираемся внести, будут небольшими и целенаправленными. Как видно на рис. 13.2, нам необходимо обновить нашу модель (этап 2А на рисунке), набор данных (2Б) и цикл обучения (2В), чтобы учесть входные и выходные данные новой модели и другие требования (не пугайтесь, если не каждый компонент этапа 2 в правой части диаграммы вам знаком, — мы рассмотрим детали каждого этапа по мере продвижения). Наконец, мы рассмотрим результаты, которые получатся при запуске нашей новой модели (этап 3 на рисунке).

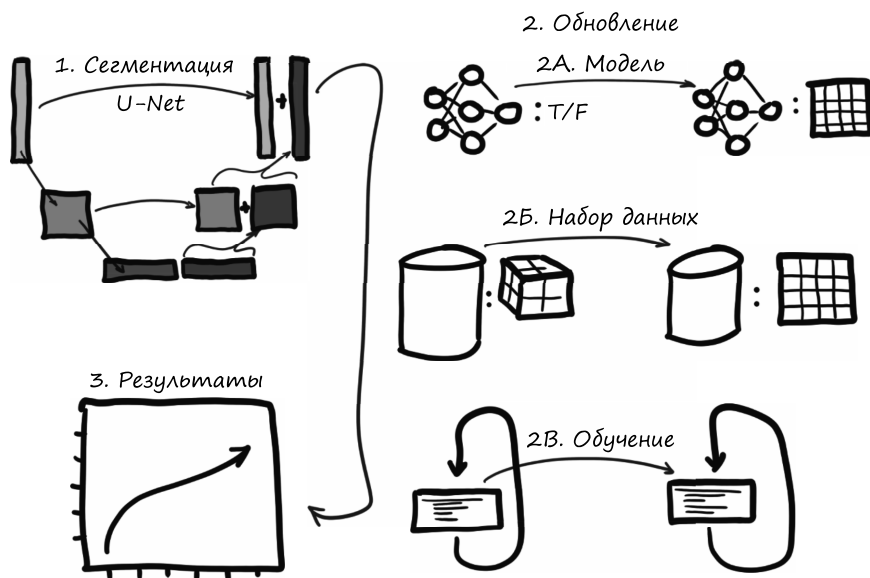


Рис. 13.2. Новая архитектура модели сегментации с моделью, набором данных и изменениями цикла обучения, которые мы реализуем

Разбив рис. 13.2 на этапы, мы можем сформулировать план на эту главу.

1. *Сегментация.* Сначала мы узнаем, как она работает с моделью U-Net, рассмотрим, что представляют собой новые компоненты модели и что с ними происходит, когда мы выполняем процесс сегментации. На рис. 13.2 — этап 1.

2. *Обновление.* Чтобы реализовать сегментацию, нам нужно изменить имеющийся код в трех основных местах, показанных в подэтапах в правой части рис. 13.2. Код будет структурно очень похож на тот, который мы разработали для классификации, но иметь некоторые различия в деталях.
 - А. *Обновление модели* (этап 2А). Мы интегрируем существующую сеть U-Net в модель сегментации. Наша модель в главе 12 выводит простую классификацию вида «истина/ложь», а в этой главе будет выводить полное изображение.
 - Б. *Изменение набора данных* (этап 2Б). Нам нужно изменить наш набор данных, чтобы в нем были не только биты КТ, но и маски узелков. Набор классификационных данных состоял из 3D-срезов, взятых вокруг узелков-кандидатов, но нам нужно будет собрать как полные срезы КТ, так и 2D-срезы для обучения и проверки сегментации.
 - В. *Адаптация цикла обучения* (этап 2В). Нам нужно адаптировать цикл обучения и добавить новое значение потери для оптимизации. Поскольку мы хотим показывать изображение результатов нашей сегментации в TensorBoard, мы также будем сохранять веса модели на диск.
3. *Результаты.* Наконец, мы посмотрим на плоды наших усилий и результаты количественной сегментации.

13.2. РАЗЛИЧНЫЕ ТИПЫ СЕГМЕНТАЦИИ

Для начала нам нужно поговорить о различных вариантах сегментации. В этом проекте мы будем использовать *семантическую* сегментацию, то есть классификацию отдельных пикселей изображения с помощью меток, похожих на те, к которым мы прибегали в задаче классификации, например «медведь», «кошка», «собака» и т. д. Если сделать все правильно, то отдельные части или области изображения будут нести свой смысл, наподобие «все эти пиксели являются частью кошки». В итоге мы получим маску метки или тепловую карту, на которой обозначены области интереса. У нас будет простая бинарная метка, истинные значения будут соответствовать узелкам-кандидатам, а ложные — неинтересной здоровой ткани. Это частично удовлетворяет нашу потребность в поиске узелков-кандидатов, которые затем нужно будет передать классификатору.

Прежде чем углубиться в детали, мы должны кратко обсудить другие подходы, которые тоже можно было бы использовать для поиска наших узелков-кандидатов. Например, при *сегментации экземпляров* отдельные интересующие объекты помечаются разными метками. Разница в том, что семантическая сегментация помечает изображение двух людей, пожимающих друг другу руки, двумя метками («человек» и «фон»), а сегментация экземпляров создаст три метки («человек1», «человек2» и «фон»), причем граница меток людей

ляжет где-то около рук. Такая сегментация помогла бы отличить «узелок 1» от «узелка 2», но вместо этого для идентификации отдельных узелков мы будем использовать группировку. Данный подход нам подойдет, поскольку узелки вряд ли будут соприкасаться или перекрываться.

Еще один подход к такого рода задачам — *обнаружение объектов*, то есть поиск нужного объекта на изображении и создание вокруг него ограничивающей рамки. В нашей задаче и сегментация экземпляров, и обнаружение объектов могли бы быть полезны, но эти методы несколько сложны в реализации, и мы не считаем, что вам стоит сейчас заниматься ими. Кроме того, для обучения моделей обнаружения объектов обычно требуется гораздо больше вычислительных ресурсов, чем требует наш подход. Если вы хотите более сложную задачу, то вам поможет статья YOLOv3, в которой больше полезной информации, чем в большинстве исследовательских работ по глубокому обучению¹. А мы пока займемся семантической сегментацией.

ПРИМЕЧАНИЕ

Рассматривая примеры кода в этой главе, мы предполагаем, что вы заглядываете в код в GitHub, в котором то же самое описано более подробно. Мы будем опускать неинтересный код или похожий на представленный в предыдущих главах, а сосредоточимся лишь на сути рассматриваемой проблемы.

13.3. СЕМАНТИЧЕСКАЯ СЕГМЕНТАЦИЯ: ПОПИКСЕЛЬНАЯ КЛАССИФИКАЦИЯ

Часто сегментация используется для ответа на вопросы вида «Где на этой картинке кот?». Очевидно, что чаще всего, как на рис. 13.3, значительная часть к коту не относится. Всегда есть стол или стена на заднем плане, клавиатура, на которой сидит кот, и т. п. Чтобы сказать: «Этот пиксель — часть кота, а вот тот, другой пиксель — часть стены», у модели должны быть принципиально другие выходные данные и внутренняя структура по сравнению с моделями классификации, с которыми мы работали до сих пор. Классификация говорит, есть ли кот на картинке, а сегментация показывает, где именно.

Если в проекте нужно различить кота на переднем или заднем плане или слева/справа, то нужна именно сегментация. Работающие с изображениями модели классификации, которые мы реализовывали до сих пор, можно рассматривать как воронки или увеличительные стекла, которые берут большой набор пикселей и фокусируют их в одну «точку» (или, точнее, в единый набор прогнозов

¹ Redmon J., Farhadi A. YOLOv3: An Incremental Improvement, <https://pjreddie.com/media/files/papers/YOLOv3.pdf>. Возможно, вам стоит ознакомиться с этой работой, когда вы дочитаете книгу.

по классам), как показано на рис. 13.4. Модели классификации дают ответы в форме «Да, где-то в этой огромной куче пикселей есть кот» или «Нет, здесь нет котов». Это отлично, когда вам все равно, где находится кот, а нужно лишь знать, что он в принципе есть.



Рис. 13.3. Результатом классификации является один или несколько бинарных флагов, а сегментация создает маску или тепловую карту

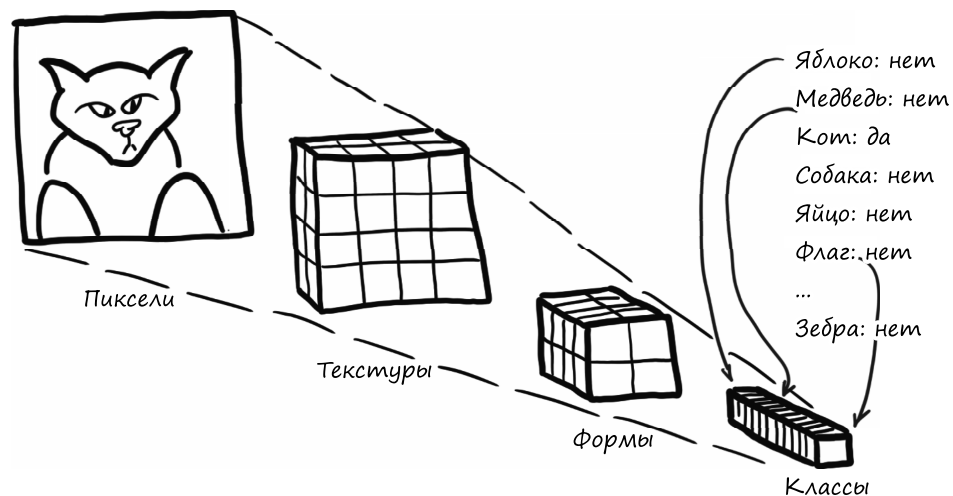


Рис. 13.4. Структура увеличительного стекла в модели классификации

Многokrатно используемые слои свертки и субдискретизации означают, что модель сначала берет необработанные пиксели для создания конкретных

и подробных детекторов таких вещей, как текстура и цвет, а затем создает детекторы концептуальных признаков более высокого уровня для, например, частей лица. В конечном итоге модель доходит до «кота» или «собаки». Из-за увеличения рецептивных полей сверток после каждого слоя понижающей дискретизации эти детекторы высокого уровня могут использовать информацию из большой области входного изображения.

К сожалению, сегментация должна производить вывод, похожий на изображение, поэтому получить единый список двоичных флагов, похожий на классификацию, не получится. Как мы помним из раздела 11.4, субдискретизация является ключом к увеличению рецептивных полей сверточных слоев и помогает свести массив пикселей, составляющих изображение, к списку классов. Обратите внимание на рис. 13.5, который повторяет рис. 11.6.

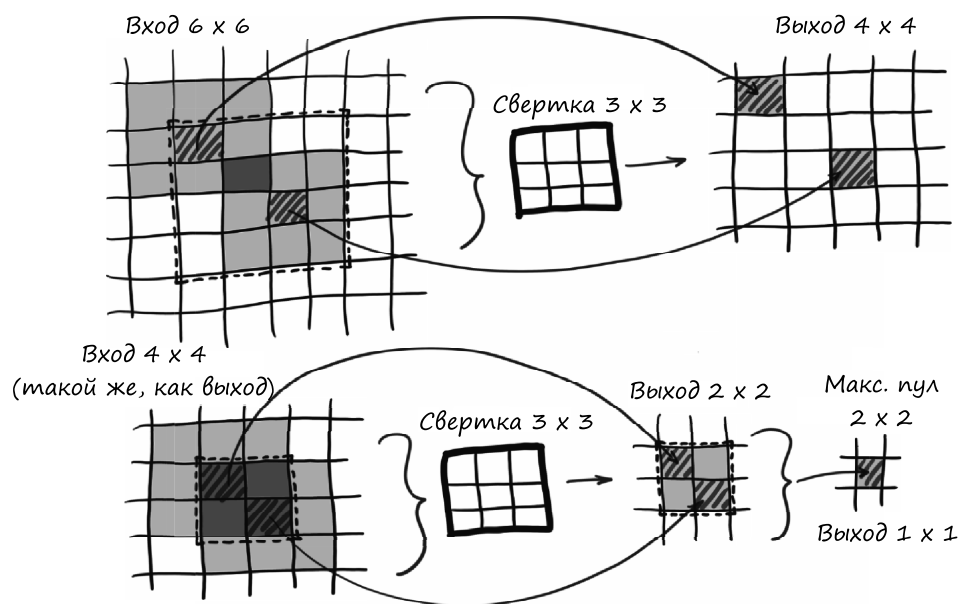


Рис. 13.5. Сверточная архитектура блока LunaModel, состоящая из двух сверток 3×3 , за которыми следует операция максимального пула. Последний пиксель имеет рецептивное поле размером 6×6

Как видите, входные данные движутся слева направо в верхнем ряду и продолжают движение в нижнем. Чтобы оценить размер рецептивного поля (области, которая формирует один пиксель в правом нижнем углу), мы можем вернуться назад. Операция максимального пула имеет вход размером 2×2 на каждый конечный выходной пиксель. Свертка 3×3 в середине нижнего ряда смотрит на один соседний пиксель (в том числе и по диагонали) в каждую сторону,

поэтому суммарное рецептивное поле сверток, приводящих к выводу 2×2 , равно 4×4 . Затем свертка 3×3 в верхней строке добавляет дополнительный пиксель контекста в каждом направлении, вследствие чего рецептивное поле одного выходного пикселя внизу справа представляет собой поле 6×6 на входе вверху слева. С понижением частоты дискретизации из максимального пула рецептивное поле следующего блока имеет вдвое большую ширину, и каждый следующий блок удваивает ее еще сильнее, сжимая размер выходных данных.

Если мы хотим, чтобы наш выход был того же размера, что и наш вход, понадобится другая архитектура модели. Простая модель, которую можно использовать для сегментации, состоит из нескольких сверточных слоев без понижающей дискретизации. При правильных отступах мы получим на выходе тот же размер, что и на входе (а это хорошо), но очень ограниченное рецептивное поле (а это плохо), поскольку эти ограничения зависят от того, насколько сильно пересекаются слои сверток.

Модель классификации использует каждый слой понижающей дискретизации, чтобы удвоить эффективный охват следующих сверток; и без этого увеличения эффективного размера поля каждый сегментированный пиксель сможет учитывать только очень локальную окрестность.

ПРИМЕЧАНИЕ

При свертках 3×3 размер рецептивного поля для простой модели сложенных сверток равен $2 \times L + 1$, где L — количество сверточных слоев.

Четыре слоя сверток 3×3 дают рецептивное поле 9×9 на выходной пиксель. Вставив максимальный пул 2×2 между второй и третьей сверткой и еще один в конце, мы увеличим рецептивное поле до...

ПРИМЕЧАНИЕ

Сначала попробуйте посчитать сами!

... 16×16 . Последняя серия conv-conv-pool имеет рецептивное поле 6×6 , но это происходит *после* первого максимального пула, что делает окончательное эффективное рецептивное поле равным 12×12 в исходном входном разрешении. Первые два сверточных слоя добавляют общую границу в два пикселя вокруг 12×12 , всего получается 16×16 .

Таким образом, остается вопрос: как улучшить рецептивное поле выходного пикселя, сохраняя при этом соотношение входных пикселей к выходным $1 : 1$? В подобных случаях обычно используется метод под названием «*повышающая дискретизация*», в котором мы берем изображение с заданным разрешением и создаем изображение с более высоким разрешением. Повышение частоты дискретизации в самом простом виде означает замену каждого пикселя блоком $N \times N$ пикселей, каждый из которых имеет тот же цвет, что и исходный входной

пиксель. Здесь появляются более интересные возможности, например линейная интерполяция и обучение развертке.

13.3.1. Архитектура U-Net

Прежде чем погрузиться в кроличью нору возможных алгоритмов повышения дискретизации, вернемся к нашей цели, которую мы поставили для этой главы. Согласно рис. 13.6 этап 1 — это знакомство с базовым алгоритмом сегментации под названием U-Net.

Архитектура U-Net — это предназначенный для сегментации дизайн нейронной сети, которая может производить попиксельные выходные данные. Как видно на рис. 13.6, схема архитектуры U-Net немного напоминает букву U, и отсюда, собственно, название. Мы также сразу видим, что структура здесь несколько сложнее, чем в основном последовательная структура знакомых нам классификаторов. Вскоре на рис. 13.7 мы рассмотрим более подробную версию архитектуры U-Net и разберемся, что делает каждый из этих компонентов. Как только мы поймем архитектуру модели, мы сможем работать над ее обучением для решения задачи сегментации.

Архитектура U-Net, показанная на рис. 13.7, в свое время стала прорывом в области сегментации изображений. Посмотрим на рисунок, а затем пройдемся по архитектуре.

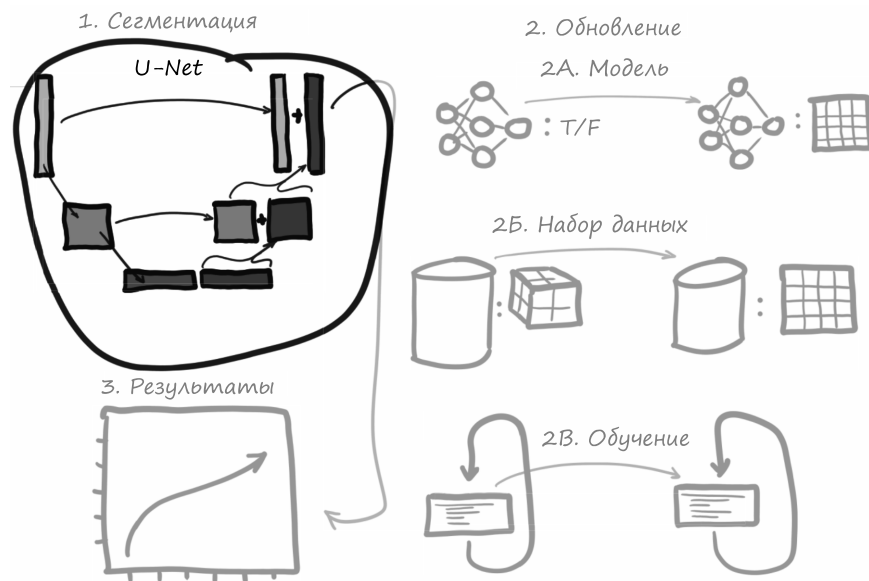


Рис. 13.6. Новая архитектура модели сегментации, с которой мы будем работать

Архитектура U-Net

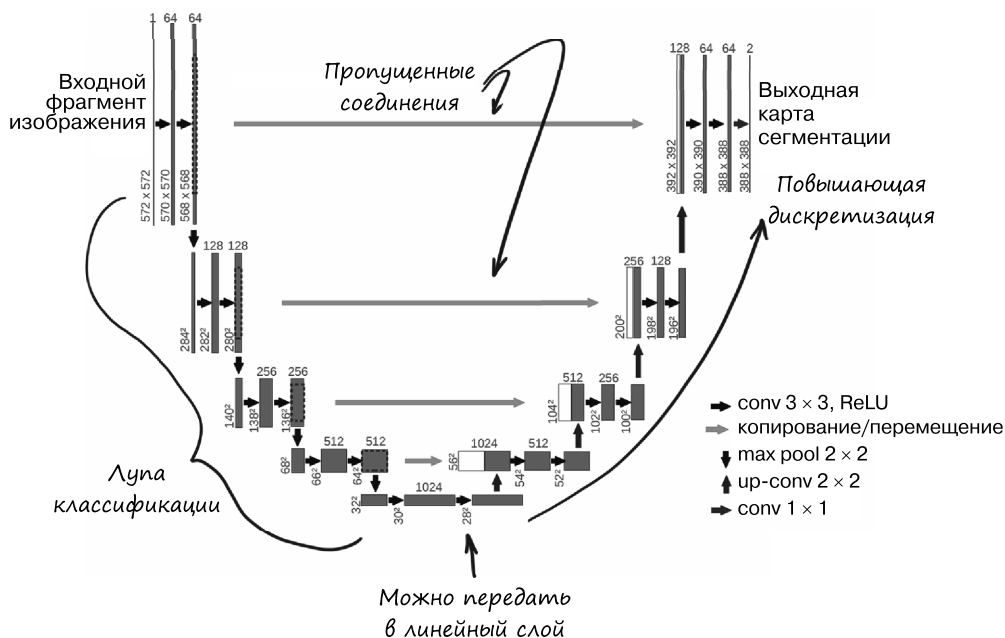


Рис. 13.7. Из статьи, посвященной U-Net, с аннотациями. Основа взята из работы Олафа Роннебергера (Olaf Ronneberger) и соавторов под названием U-Net: Convolutional Networks for Biomedical Image Segmentation, которую можно найти на <https://arxiv.org/abs/1505.04597> и <https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net>

На этой диаграмме прямоугольниками обозначены промежуточные результаты, а стрелками — операции между ними. U-образная форма архитектуры образуется множеством разрешений, с которыми работает сеть.

В верхнем ряду — полное разрешение (у нас 512×512), в нижнем — вдвое меньше и т. д. Данные проходят сверху слева к центру через серию сверток и масштабирования — похожее мы уже видели в классификаторах и подробно рассматривали в главе 8. Затем мы снова поднимаемся вверх, используя развертку, чтобы вернуться к полному разрешению. В отличие от оригинальной U-Net, мы будем делать отступы, чтобы не терять пиксели по краям, поэтому разрешение слева и справа получится одинаковым.

В ранних проектах сетей уже присутствовала эта U-образная форма, и люди пытались использовать ее для решения проблемы ограниченности рецептивного поля полных сверточных сетей. Чтобы побороть ограничения, вводился дизайн, в котором фокусирующие фрагменты копировались, инвертировались и добавлялись в сеть классификации изображений для создания симметричной

модели, которая переходит от мелких деталей к широкому рецептивному полю и обратно к мелким деталям.

Однако в этих более ранних проектах у таких сетей были проблемы со сходимостью, скорее всего, из-за потери пространственной информации во время субдискретизации. Как только информация превращается в большое количество очень уменьшенных изображений, точное местоположение границ объектов становится сложнее закодировать и воспроизвести. Чтобы решить эту проблему, авторы U-Net добавили пропущенные соединения, которые мы видим в центре рис. 13.7. Впервые мы затронули понятие пропущенных соединений в главе 8, но здесь они используются не так, как в архитектуре ResNet. В U-Net пропущенные соединения связывают входы на пути понижения дискретизации с соответствующими слоями на пути повышения дискретизации. Эти слои получают в качестве входных данных как результаты повышения частоты дискретизации слоев широкого рецептивного поля из нижних слоев U, так и выходные данные более ранних слоев с мелкими деталями через мостовые соединения вида «копировать и обрезать». Это ключевое нововведение U-Net (которое, что интересно, появилось раньше ResNet).

В результате окончательные слои детализации берут лучшее из обоих источников данных. У них есть информация о более широком контексте, окружающем непосредственную область, а также подробные данные из первого набора слоев с полным разрешением.

Слой $\text{conv } 1 \times 1$ в крайнем правом углу сети изменяет количество каналов с 64 на 2 (в исходной статье было два выходных канала, в нашем случае — один). Это чем-то похоже на полносвязный слой, который мы применяли в сети классификации, но работающий попиксельно и по каналам. Как следствие, можно преобразовать количество фильтров, использованных на последнем этапе повышающей дискретизации, в количество необходимых выходных классов.

13.4. ОБНОВЛЕНИЕ МОДЕЛИ СЕГМЕНТАЦИИ

Пришло время перейти к этапу 2А на рис. 13.8. Мы достаточно поговорили о сегментации и истории U-Net, теперь пришла пора обновить код, начиная с модели. Вместо того чтобы просто выводить бинарную классификацию, которая дает вывод в виде «истина» или «ложь», мы реализуем U-Net, чтобы получить модель, способную выводить значение вероятности для каждого пикселя, то есть выполнять сегментацию.

Вместо того чтобы реализовывать пользовательскую модель сегментации U-Net с нуля, мы задействуем существующую реализацию из репозитория с открытым исходным кодом в GitHub.

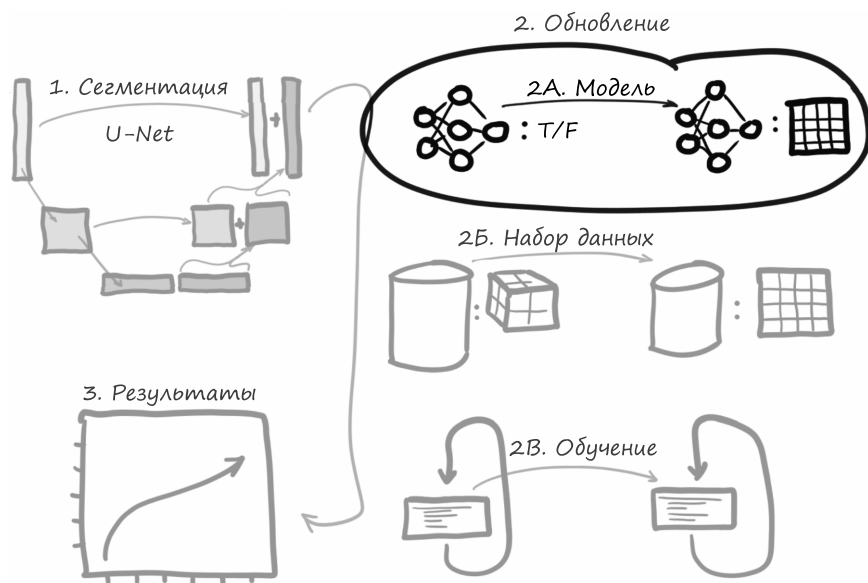


Рис. 13.8. План этого раздела — изменения, необходимые для нашей модели сегментации

Реализация U-Net по адресу <https://github.com/jvanvugt/pytorch-unet> хорошо отвечает нашим потребностям¹. Она лицензирована MIT (права защищены в 2018 году, Joris), содержится в одном файле и позволяет настраивать ряд параметров. Этот файл можно найти в нашем репозитории кода по адресу `util/unet.py` вместе со ссылкой на исходный репозиторий и полным текстом используемой лицензии.

ПРИМЕЧАНИЕ

Хотя для личных проектов это не так важно, но обычно вы должны знать условия лицензий программного обеспечения с открытым исходным кодом, которое вы используете для проекта. Лицензия MIT — одна из самых либеральных лицензий с открытым исходным кодом, но все же некоторые требования в ней есть! Кроме того, имейте в виду, что авторы сохраняют авторские права, даже если они публикуют свою работу на общедоступном форуме (и даже в GitHub), и если лицензия не оформляется явным образом, то это не означает, что работа находится в общественном достоянии. Наоборот! Это говорит о том, что у вас нет лицензии на использование кода, равно как и права на копирование книги, которую вы взяли в библиотеке.

¹ Представленная здесь реализация отличается от официальной статьи тем, что для понижения частоты дискретизации используется объединение средних значений вместо объединения максимальных значений. В самой последней версии в GitHub используется максимальный пул.

Мы уделим некоторое время изучению кода и, основываясь на знаниях, имеющихся у нас к данному моменту, определим строительные блоки архитектуры в коде. Сможете ли вы сами найти пропущенные соединения? В качестве полезного упражнения рекомендуем вам, глядя на код, начертить диаграмму устройства модели.

Теперь, когда мы нашли реализацию U-Net, отвечающую нашим требованиям, нам нужно адаптировать ее под наши нужды. В целом полезно находить возможности применять что-то готовое. Важно иметь представление о том, какие модели существуют, как они реализуются и обучаются и можно ли извлечь из них какие-либо части и применить их к проекту, над которым мы работаем в любой момент. Этот навык приходит со временем и опытом, но уже сейчас вы можете начать создавать свой набор инструментов.

13.4.1. Адаптация готовой модели к нашему проекту

Сейчас мы внесем некоторые изменения в классическую U-Net, разъясняя, почему и зачем. В качестве полезного упражнения вы можете сравнить результаты работы *оригинальной* модели и модели после настройки и вдобавок оценить влияние каждого изменения, удаляя их по одному (это также называется *исследованием абляции*).

Во-первых, мы будем передавать входные данные через пакетную нормализацию. Тогда нам не придется самостоятельно нормализовать данные в наборе данных; и, что более важно, мы получим статистику нормализации (среднее значение и стандартное отклонение), рассчитанную по отдельным пакетам. Это означает, что если пакет окажется *скучным* (не будет содержать нужных нам данных), то будет масштабироваться сильнее. Случайный выбор элементов данных в пакетах в каждую эпоху сводит к минимуму вероятность того, что скучный элемент окажется в полностью скучном пакете, и, следовательно, такие элементы будут рассматриваться чрезмерно внимательно.

Во-вторых, поскольку выходные значения не ограничены, мы должны пропустить выходные данные через слой `nn.Sigmoid`, чтобы ограничить их диапазоном $[0, 1]$. В-третьих, мы уменьшим общую глубину и количество фильтров, которые модель будет применять. Немного забегаая вперед, отметим, что возможности модели, использующей стандартные параметры, намного превосходят размер нашего набора данных. Это значит, что мы вряд ли найдем предварительно обученную модель, которая будет точно соответствовать нашим потребностям. Наконец, хотя это и не модификация, важно отметить, что выходные данные получаются одноканальными, где каждый пиксель вывода содержит оценку вероятности того, что он является частью узелка.

Эту оболочку U-Net довольно просто реализовать в виде модели с тремя атрибутами: по одному для двух функций, которые мы хотим добавить, и один для

самой U-Net, которую мы можем рассматривать как готовый модуль. Мы также передадим любые полученные именованные аргументы в конструктор U-Net (листинг 13.1).

Листинг 13.1. model.py:17, класс UNetWrapper

```
class UNetWrapper(nn.Module):
    def __init__(self, **kwargs):
        super().__init__()

        self.input_batchnorm = nn.BatchNorm2d(kwargs['in_channels'])
        self.unet = UNet(**kwargs)
        self.final = nn.Sigmoid()

        self._init_weights()
```

kwargs — это словарь, содержащий все ключевые слова-аргументы, передаваемые конструктору

BatchNorm2d ожидает, что мы укажем количество каналов, которое берем из именованного аргумента

Как и для классификатора в главе 11, мы инициализируем веса по-своему. Функция та же самая, поэтому ее код мы уже видели

U-Net: изменение небольшое, но именно оно делает всю работу

Метод `forward` — это простая последовательность. Мы могли бы использовать экземпляр `nn.Sequential`, как уже делали в главе 8, но здесь для ясности кода и трассировки стека выберем более явную реализацию¹ (листинг 13.2).

Листинг 13.2. model.py:50, UNetWrapper.forward

```
def forward(self, input_batch):
    bn_output = self.input_batchnorm(input_batch)
    un_output = self.unet(bn_output)
    fn_output = self.final(un_output)
    return fn_output
```

Обратите внимание: здесь мы используем `nn.BatchNorm2d`. Это связано с тем, что U-Net — это двумерная модель сегментации. Мы могли бы адаптировать реализацию для использования 3D-сверток, чтобы передавать информацию между срезами.

Затраты памяти в прямолинейной реализации окажутся значительно больше, и нам пришлось бы нарезать компьютерную томографию. Кроме того, тот факт, что расстояние между пикселями в направлении *Z* намного больше, чем по другим направлениям, снижает вероятность присутствия узелков на многих срезах. Из-за этого полностью трехмерный подход для наших целей оказывается менее привлекательным. Вместо этого мы адаптируем наши 3D-данные для сегментации по срезам, предоставляя соседние срезы для контекста (например, определить, что яркая выпуклость действительно является кровеносным сосудом, будет проще, если известны соседние срезы). Поскольку данные у нас имеют

¹ Вряд ли наш код выдаст какие-либо исключения, не так ли?

формат 2D, для представления смежных срезов мы будем использовать каналы. Наша трактовка третьего измерения аналогична тому, как мы применяли полносвязную модель к изображениям в главе 7: модели придется заново научиться распознавать соседние пиксели, которые мы в данный момент отбрасываем наряду с направлением вдоль оси. Впрочем, для модели это не составит труда, особенно учитывая, насколько большой контекст она получает в виде среза по сравнению с размером самого узелка.

13.5. МОДИФИКАЦИЯ НАБОРА ДАННЫХ ДЛЯ СЕГМЕНТАЦИИ

Исходные данные для этой главы остаются все теми же: мы используем КТ-сканы и аннотации к ним. Но наша модель ожидает ввод в другой форме и произведет другой вывод. На этапе 2Б на рис. 13.9 мы намекаем, что раньше в наборе были 3D-данные, а теперь нам нужно создать 2D-данные.

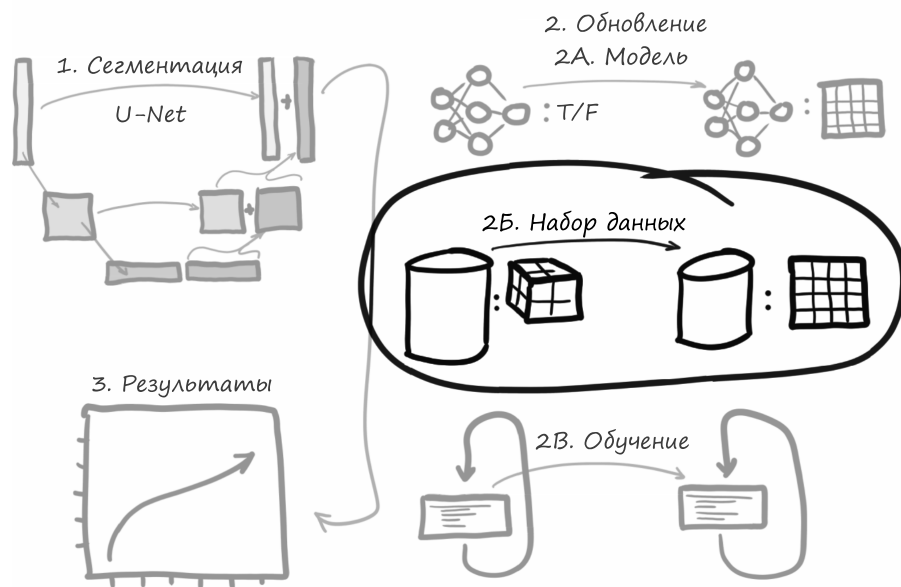


Рис. 13.9. План этого раздела — изменения в наборе данных для сегментации

В первой реализации U-Net не использовались дополненные свертки; это значит, что, хотя выходная карта сегментации была меньше, чем входная, каждый пиксель этого вывода имел полностью заполненное рецептивное поле.

Ни один из входных пикселей не был заполнен, сфабрикован или иным образом произведен неполным. Таким образом, вывод оригинальной U-Net будет идеально мозаичным, поэтому модель можно использовать с изображениями любого размера (кроме краев входного изображения, где часть контекста будет отсутствовать по определению).

Поскольку мы используем один и тот же подход к задаче с точностью до пикселя, возникает две проблемы. Первая связана со взаимодействием между сверткой и субдискретизацией, а вторая — с тем, что данные у нас трехмерные.

13.5.1. Особые требования U-Net к размеру входных данных

Первая проблема заключается в том, что размеры входных и выходных фрагментов данных в U-Net очень специфичны. Чтобы потери в два пикселя на свертку равномерно выровнялись до и после понижения дискретизации (особенно если учесть дальнейшее сжатие свертки при более низком разрешении), сети нужны строго определенные входные размеры. В документе U-Net использовались фрагменты изображений размером 512×512 , что дает выходные карты размером 388×388 . Входные изображения должны быть больше, чем наши срезы 512×512 , а выходные данные оказываются чуть меньше! Это означало бы, что узелки вблизи края среза КТ вообще не будут сегментированы. Такой подход хорошо работает при работе с очень большими изображениями, но для нашей задачи неидеален.

Мы решим данную проблему, установив для флага `padding` конструктора U-Net значение `True`. Это будет означать, что мы можем использовать входные изображения любого размера и вывод получим того же размера. Возможны потери точности вблизи краев изображения, поскольку рецептивное поле пикселей вблизи края будет включать искусственно дополненные области, но это компромисс, с которым придется смириться.

13.5.2. Компромиссы U-Net при работе с 3D- и 2D-данными

Вторая проблема заключается в том, что наши 3D-данные не совсем совпадают с ожидаемыми U-Net двумерными входными данными. Просто взять наше изображение размером $512 \times 512 \times 128$ и передать его в преобразованный в 3D класс U-Net не получится, поскольку на это не хватит памяти графического процессора. Каждое изображение имеет размеры $2^9 \times 2^9 \times 2^7$, по 2^2 байта на воксель. Первый уровень U-Net — это 64 канала, или 2^6 . Это показатель степени $9 + 9 + 7 + 2 + 6 = 33$, или 8 Гбайт, *только для первого сверточного слоя*. У нас два

сверточных слоя (16 Гбайт), а затем каждое понижение разрешения уменьшает разрешение вдвое, но удваивает каналы, то есть еще 2 Гбайт для каждого слоя после первого понижения разрешения (помните, что уменьшение разрешения вдвое приводит к снижению объема данных в восемь раз, поскольку мы работаем с 3D-данными). В итоге мы дошли до 20 Гбайт еще до того, как достигли второго понижения разрешения.

ПРИМЕЧАНИЕ

Существует множество хитрых и новаторских способов обойти эти проблемы, и мы ни в коем случае не утверждаем, что наш подход — единственный рабочий¹. Мы считаем, что он является одним из самых простых и для уровня проекта в данной книге вполне годится. Мы стараемся применять простые инструменты, чтобы лучше сосредоточиться на фундаментальных концепциях. К хитростям можно прибегнуть позже, как только вы освоите основы.

Поэтому вместо попыток делать что-то в 3D мы будем рассматривать каждый срез как задачу двумерной сегментации, а в качестве контекста третьего измерения станем передавать соседние срезы как отдельные каналы. Вместо традиционных красных, зеленых и синих каналов, используемых в фотографиях, нашими каналами будут «на два среза выше», «на один срез выше», «сегментируемый срез», «на один срез ниже» и т. д.

Но и этот подход не лишен компромиссов. Мы теряем прямую пространственную связь между срезами, когда они передаются в виде каналов, поскольку все каналы будут линейно объединены ядрами свертки без указания того, с какой стороны и на каком расстоянии находятся срезы. Мы также теряем более широкое рецептивное поле в измерении глубины, которое было бы получено при настоящей 3D-сегментации с понижением частоты дискретизации. Поскольку срезы КТ часто толще, чем разрешение по строкам и столбцам, мы получаем более широкое изображение, чем кажется на первый взгляд, и этого должно быть достаточно, учитывая, что узелки обычно охватывают ограниченное количество срезов.

Еще один момент, который следует учитывать и в текущем, и в трехмерном подходе, заключается в том, что теперь мы игнорируем точную толщину среза. Наша модель должна в конечном итоге стать устойчивой, так как получает данные с разными расстояниями между срезами.

В целом не существует простой блок-схемы или эмпирического правила, в котором можно было бы найти готовые ответы на вопросы о том, на какие

¹ Например, ознакомьтесь с работой: *Nikolov S. et al. Deep Learning to Achieve Clinically Applicable Segmentation of Head and Neck Anatomy for Radiotherapy*, <https://arxiv.org/pdf/1809.04430.pdf>.

компромиссы можно пойти и не слишком ли данный набор компромиссов обременителен. Важнее всего тщательно проводить эксперименты и систематически проверять гипотезы одну за другой. Только так можно определить, какие изменения и методы лучше всего подходят для рассматриваемой проблемы. Всегда есть соблазн внести сразу множество изменений и получить идеальный результат, однако *не поддавайтесь этому импульсу*.

Еще раз: *никогда не тестируйте несколько модификаций одновременно*. Слишком велика вероятность того, что одно из изменений будет плохо взаимодействовать с другим и у вас не будет явных доказательств для того, чтобы понять, заслуживает ли то либо иное изменение дальнейшего изучения. С учетом сказанного начнем создавать набор данных для сегментации.

13.5.3. Формирование достоверных данных

Первая проблема заключается в несоответствии между обучающими данными, помеченными людьми, и фактическим результатом, который мы хотим получить от нашей модели. У нас есть аннотированные точки, а нужна маска для каждого вокселя, указывающая, является ли данный воксель частью узелка. Нам придется построить эту маску самостоятельно на основе имеющихся у нас данных, а затем выполнить ручную проверку, чтобы убедиться, что подпрограмма, создающая маску, работает хорошо.

Проверять вручную созданные эвристики на большом количестве данных может быть сложно. Мы не будем реализовывать нечто излишне сложное, чтобы проверить, правильно ли каждый узелок обрабатывался нашей эвристикой. Если бы у нас было больше ресурсов, то мы могли бы нанять кого-то для создания и/или проверки результатов вручную. Но поскольку у нас не очень хорошо финансируемое предприятие, мы выполним проверку нескольких образцов и с помощью очень простого вопроса: «Выглядит ли результат приемлемым?»

С этой целью мы будем разрабатывать подходы и API так, чтобы упростить исследование промежуточных шагов алгоритмов. В результате есть шанс получить несколько неказистые вызовы функций, которые будут возвращать огромные кортежи промежуточных значений, но возможность легко получить результаты и вывести оправдывает эту неказистость.

Ограничительные рамки

Начнем с преобразования местоположений имеющихся узелков в ограничивающие рамки, которые охватывают весь узелок (обратите внимание, что мы сделаем это только для *реальных узелков*). Предположив, что точка расположения узелка находится примерно в его центре, мы можем отступить наружу

от этой точки во всех трех измерениях, пока не наткнемся на воксели с низкой плотностью. Это будет свидетельствовать о том, что мы дошли до нормальной легочной ткани (которая в основном заполнена лейкоцитами). Данный алгоритм изображен на рис. 13.10.

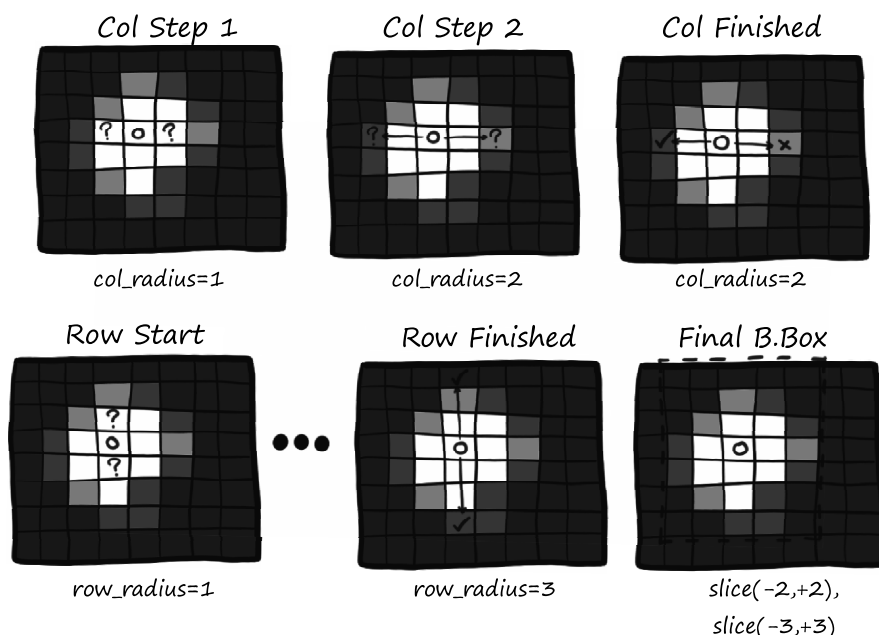


Рис. 13.10. Алгоритм определения ограничивающей рамки вокруг узелка

Мы начинаем поиск (точка O на рисунке) в аннотированном вокселе, считая его центром узелка. Затем проверяем плотность вокселей, прилегающих к началу координат на оси столбца, отмеченной знаком вопроса (?). Поскольку оба исследованных вокселя содержат плотную ткань, отмеченную более светлым цветом, мы продолжаем поиск. Увеличив расстояния поиска в столбце до 2, мы обнаруживаем, что левый воксель имеет плотность ниже заданного порога, поэтому останавливаем поиск.

Затем мы выполняем тот же поиск в направлении строки. Снова начинаем с начала и на этот раз ищем вверх и вниз. Когда расстояние поиска стало равным 3, мы нашли воксели низкой плотности сверху и снизу. Но для остановки хватило бы и одного вокселя.

Поиск в третьем измерении опустим. Наша последняя ограничительная рамка имеет пять вокселей в ширину и семь в высоту. Вот как это выглядит в коде (листинг 13.3).

Листинг 13.3. dsets.py:131, Ct.buildAnnotationMask

```

center_irc = xyz2irc(
    candidateInfo_tup.center_xyz, ← кандидатInfo_tup здесь такой же,
    self.origin_xyz,               как мы видели ранее: он возвращается
    self.vxSize_xyz,              функцией getCandidateInfoList
    self.direction_a,
)
ci = int(center_irc.index) ← Получаем индексы центральных
cr = int(center_irc.row)   вокселей, отправную точку
cc = int(center_irc.col)

index_radius = 2
try:
    while self.hu_a[ci + index_radius, cr, cc] > threshold_hu and \
        self.hu_a[ci - index_radius, cr, cc] > threshold_hu: ← Поиск,
        index_radius += 1                                       описанный
                                                                ранее
except IndexError: ← Страховочная сетка для индексации
    index_radius -= 1   за пределами тензора

```

Сначала мы получаем данные о центральной точке, а затем выполняем поиск в цикле `while`. Немного осложняет дело тот факт, что поиск может выйти за границы тензора. Нас это не слишком беспокоит, и мы ленивы, поэтому просто перехватываем исключение индекса¹.

Обратите внимание, что мы прекращаем увеличивать очень приблизительные значения `radius` после того, как плотность падает ниже порогового значения, поэтому ограничивающая рамка должна содержать одновоксельную границу ткани с низкой плотностью (по крайней мере с одной стороны, поскольку узелки могут примыкать к плотной ткани и мы должны прекратить поиск в обоих направлениях, когда наткнемся на воздух с любой стороны). Мы проверяем `center_index + index_radius`, и `center_index - index_radius`, в связи с чем эта одновоксельная граница будет существовать только на краю, ближайшем к нашему местоположению узелка. Вот почему нам нужно, чтобы эти места были по возможности центрированы. Поскольку некоторые узелки примыкают к границе между легким и более плотной тканью, такой как мышца или кость, мы не можем сдвигать границу в каждом направлении независимо, иначе некоторые края окажутся невероятно далеко от самого узелка.

Затем мы повторяем тот же процесс расширения радиуса с `row_radius` и `col_radius` (для краткости данный код опущен). Как только это будет сделано, мы можем установить рамку в нашем массиве равной `True` (через мгновение мы увидим определение `boundingBox_ary`, что неудивительно).

Запишем все это в функцию. В цикле перебираем все узелки. Для каждого узелка мы выполняем поиск, показанный ранее (который мы исключаем из

¹ Ошибка здесь в том, что переход в 0 останется незамеченным. Для нас это не имеет большого значения. В качестве упражнения реализуйте правильную проверку границ.

листинга 13.4). Затем в булевом тензоре `boundingBox_a` мы отмечаем найденную ограничивающую рамку.

По окончании цикла мы выполняем очистку, а именно удалим пересечение между маской ограничительной рамки и тканью, плотность которой превышает пороговое значение -700 HU (или $0,3$ г/см³). Таким образом мы обрежем уголки рамок узелков (по крайней мере, тех, которые не встроены в стенку легкого), и они будут чуть лучше соответствовать контурам узелка (листинг 13.4).

Листинг 13.4. `dsets.py:127, Ct.buildAnnotationMask`

```
def buildAnnotationMask(self, positiveInfo_list, threshold_hu = -700):
    boundingBox_a = np.zeros_like(self.hu_a, dtype=np.bool)

    for candidateInfo_tup in positiveInfo_list:
        # ... строка 169
        boundingBox_a[
            ci - index_radius: ci + index_radius + 1,
            cr - row_radius: cr + row_radius + 1,
            cc - col_radius: cc + col_radius + 1] = True

    mask_a = boundingBox_a & (self.hu_a > threshold_hu)
    return mask_a
```

Начинается с заполненного значениями False тензора того же размера, что и КТ

В цикле перебираем узелки. Напомним, что мы смотрим только на узелки, поэтому переменная называется `positiveInfo_list`

После того как радиус узелка станет известен (сам поиск уже кончился), отмечаем ограничивающую рамку

Ограничиваем распространение рамки с помощью порога плотности

Обратимся к рис. 13.11 и посмотрим, как эти маски выглядят на практике. Дополнительные полноцветные изображения можно найти в документе `p2ch13_explore_data.ipynb`.

Маска узла в правом нижнем углу демонстрирует недостаток нашего подхода с прямоугольной ограничивающей рамкой, так как в нее попала часть стенки легкого. Определенно мы могли бы это исправить, но, поскольку мы еще не уверены, что это стоит затрат времени и внимания, пока оставим все как есть¹. Теперь нужно добавить данную маску в класс КТ.

Создание маски в момент инициализации КТ

Теперь, когда мы можем взять список кортежей с информацией об узелках и превратить их в бинарную маску, встроим эти маски в наш объект КТ. Сначала мы сформируем из кандидатов список, содержащий только узелки, а затем с его помощью создадим аннотацию. Наконец, мы возьмем множество уникальных индексов массива, имеющих хотя бы один воксель маски узелка. Это будет нужно для формирования данных, которые мы используем для проверки (листинг 13.5).

¹ Исправление этой проблемы не очень поможет вам в изучении PyTorch.

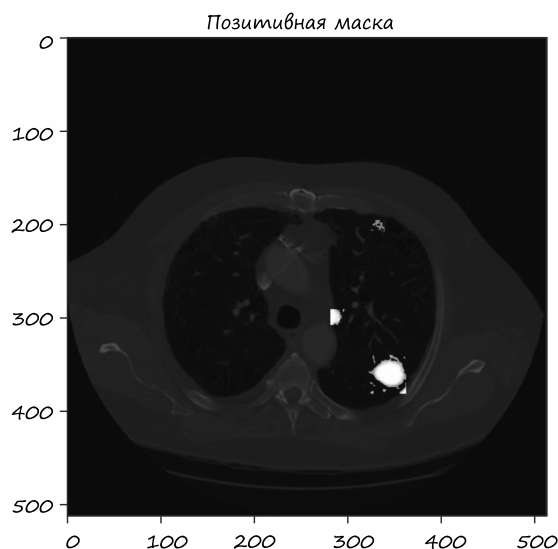


Рис. 13.11. Три узелка из `ct.positive_mask`, выделенные белым цветом

Листинг 13.5. `dsets.py:99, Ct.__init__`

```
def __init__(self, series_uid):
    # ... строка 116
    candidateInfo_list = getCandidateInfoDict()[self.series_uid]

    self.positiveInfo_list = [
        candidate_tup
        for candidate_tup in candidateInfo_list
        if candidate_tup.isNodule_bool
    ]
    self.positive_mask = self.buildAnnotationMask(self.positiveInfo_list)
    self.positive_indexes = (self.positive_mask.sum(axis=(1,2))
                             .nonzero()[0].tolist())
```

Берем только узелки

Мы получаем одномерный вектор
(по срезам) с количеством вокселей,
помеченных в маске в каждом срезе

Берем индексы срезов маски с ненулевым
количеством узелков и превращаем их в список

Внимательный взгляд мог бы заметить функцию `getCandidateInfoDict`. Ее определение нас уже не удивит, ведь это просто переформулировка той же информации, которая содержится в функции `getCandidateInfoList`, но предварительно сгруппированная по `series_uid` (листинг 13.6).

Листинг 13.6. `dsets.py:87`

```
@functools.lru_cache(1)
def getCandidateInfoDict(requireOnDisk_bool=True):
    candidateInfo_list = getCandidateInfoList(requireOnDisk_bool)
    candidateInfo_dict = {}

    for candidateInfo_tup in candidateInfo_list:
```

Это может быть полезно, чтобы метод `Ct.__init__`
не стал узким местом в производительности

```

candidateInfo_dict.setdefault(candidateInfo_tup.series_uid,
                               []).append(candidateInfo_tup)

return candidateInfo_dict

```

← Берем список кандидатов по UID из словаря, по умолчанию создавая новый пустой список, если ничего не найдется. Затем добавляем к нему существующий candidateInfo_tup

Кэширование участков маски вместе с КТ

В предыдущих главах мы кэшировали фрагменты КТ вокруг кандидатов на узелки, поскольку не хотели читать и анализировать все данные КТ каждый раз, когда нам нужен был лишь фрагмент. Мы хотим сделать то же самое с `positive_mask`, так что нам нужно еще и вернуть ее из функции `Ct.getRawCandidate`. Для этого нужны лишь одна дополнительная строка кода и изменение оператора `return` (листинг 13.7).

Листинг 13.7. `dsets.py:178, Ct.getRawCandidate`

```

def getRawCandidate(self, center_xyz, width_irc):
    center_irc = xyz2irc(center_xyz, self.origin_xyz, self.vxSize_xyz,
                        self.direction_a)

    slice_list = []
    # ... строка 203
    ct_chunk = self.hu_a[tuple(slice_list)]
    pos_chunk = self.positive_mask[tuple(slice_list)]    Новая строка

    return ct_chunk, pos_chunk, center_irc    ← Новое возвращаемое значение

```

Данные, в свою очередь, кэшируются на диск функцией `getCtRawCandidate`, которая открывает КТ, получает указанного необработанного кандидата, включая маску узелка, и обрезает значения КТ перед возвратом фрагмента КТ, маски и информации о центре (листинг 13.8).

Листинг 13.8. `dsets.py:212`

```

@raw_cache.memoize(typed=True)
def getCtRawCandidate(series_uid, center_xyz, width_irc):
    ct = getCt(series_uid)
    ct_chunk, pos_chunk, center_irc = ct.getRawCandidate(center_xyz,
                                                         width_irc)
    ct_chunk.clip(-1000, 1000, ct_chunk)
    return ct_chunk, pos_chunk, center_irc

```

Сценарий `prerpcache` предварительно вычисляет и сохраняет для нас все эти значения, помогая ускорить обучение.

Очистка данных аннотации

Еще одна вещь, которую нужно сделать в этой главе, — улучшить скрининг данных аннотаций. Выяснилось, что в файле `candidates.csv` некоторые кандидаты

указаны несколько раз. Что еще более интересно, эти записи не являются точными копиями друг друга. Создается впечатление, будто исходные, созданные людьми аннотации не были достаточно хорошо очищены перед записью в файл. Это могут быть аннотации одного и того же узелка на разных срезах, что, возможно, даже полезно для нашего классификатора.

Здесь мы сделаем щедрый жест и предоставим очищенный файл `annotation.csv`. Чтобы понять, как он возник, вам нужно знать, что набор данных LUNA получен из другого набора данных под названием Lung Image Database Consortium (LIDC-IDRI)¹ и включает подробные аннотации от нескольких рентгенологов. Мы уже проделали всю работу, чтобы получить исходные аннотации LIDC, извлекли узлы, удалили копии и сохранили результат в файл `/data/part2/luna/annotations_with_malignancy.csv`.

С помощью этого файла мы можем обновить нашу функцию `getCandidateInfoList`, чтобы получить узелки из нового файла аннотаций. Во-первых, мы перебираем новые аннотации для настоящих узелков. Используя считыватель CSV², мы должны преобразовать данные в соответствующие типы, чтобы их можно было добавить в структуру `CandidateInfoTuple` (листинг 13.9).

Листинг 13.9. `dsets.py:43, def getCandidateInfoList`

```
candidateInfo_list = []
with open('data/part2/luna/annotations_with_malignancy.csv', "r") as f:
    for row in list(csv.reader(f))[1:]:
        series_uid = row[0]
        annotationCenter_xyz = tuple([float(x) for x in row[1:4]])
        annotationDiameter_mm = float(row[4])
        isMal_bool = {'False': False, 'True': True}[row[5]]

        candidateInfo_list.append(
            CandidateInfoTuple(
                True,
                True,
                isMal_bool,
                annotationDiameter_mm,
                series_uid,
                annotationCenter_xyz,
            )
        )
```

← Для каждой строки в файле аннотаций, соответствующей узелку...

← ...мы добавляем запись в список

← isNodule_bool

← hasAnnotation_bool

¹ Armato S. G. 3rd et al. The Lung Image Database Consortium (LIDC) and Image Database Resource Initiative (IDRI): A Completed Reference Database of Lung Nodules on CT Scans, Medical Physics, 38, вып. 2, 2011, 915–931, <https://pubmed.ncbi.nlm.nih.gov/21452728/>. См. также: Брюс Вендт (Bruce Vendt), LIDC-IDRI, Cancer Imaging Archive, <http://mng.bz/mBO4>.

² Если вы часто занимаетесь этим, то библиотека `pandas`, которая вышла в версии 1.0 в 2020 году, — отличный инструмент. Здесь мы используем инструмент чтения CSV из стандартной библиотеки Python.

Как и раньше, перебираем кандидатов из файла `candidates.csv`, однако на сей раз используем только не являющиеся узелками точки. Поскольку это не узелки, информация будет заполнена значением `False` и `0` (листинг 13.10).

Листинг 13.10. `dsets.py:62, def getCandidateInfoList`

```
with open('data/part2/luna/candidates.csv', "r") as f:
    for row in list(csv.reader(f))[1:]:  ← Для каждой строки в файле кандидатов...
        series_uid = row[0]
        # ... строка 72
        if not isNodule_bool:  ← ...не являющихся узелками....
            candidateInfo_list.append(  ← ...добавляем запись
                CandidateInfoTuple(
                    False,  ← isNodule_bool
                    False,  ← isMal_bool
                    False,  ← hasAnnotation_bool
                    0.0,
                    series_uid,
                    candidateCenter_xyz,
                )
            )
```

Помимо добавления флагов `hasAnnotation_bool` и `isMal_bool` (которые мы не будем применять в этой главе), новые аннотации будут вставляться и использоваться так же, как и старые.

ПРИМЕЧАНИЕ

Вы можете спросить, почему мы не упоминали LIDC ранее. Как оказалось, в LIDC есть большое количество инструментов, которые уже созданы для базового набора данных, специфичного для LIDC. В PyLIDC можно даже получить готовые маски. Этот инструментарий представляет несколько нереалистичную картину того, какую поддержку может иметь данный набор данных, поскольку LIDC поддерживается чересчур хорошо. То, что мы сделали с данными LUNA, происходит в жизни чаще и лучше подходит для обучения, так как мы тратим время на работу с необработанными данными, а не на изучение API, который кто-то придумал за нас.

13.5.4. Реализация `Luna2dSegmentationDataset`

В этой главе мы собираемся применить не такой подход к разделению данных на обучающие и проверочные, как в предыдущих главах. У нас будет два класса: один будет действовать как общий базовый класс для проверочных данных, а другой выступит подклассом базы для обучающего набора с рандомизацией и обрезанной выборкой.

Этот подход в некоторых отношениях кажется сложнее (например, классы не полностью инкапсулированы), однако на самом деле он упрощает логику выбора случайных обучающих данных. Кроме того, становится предельно

ясно, какие ветки кода влияют как на обучение, так и на проверку, а какие используются только для обучения. Без этого часть логики может стать глубоко вложенной или переплетенной таким образом, что ее будет трудно понять. Это важно, поскольку наши обучающие данные будут значительно отличаться от проверочных данных!

ПРИМЕЧАНИЕ

Можно использовать и другие схемы классов; например, мы рассматривали возможность создания двух совершенно отдельных подклассов набора данных. Мы опираемся на стандартные принципы разработки программного обеспечения, поэтому старайтесь поддерживать относительно простую структуру, не дублируйте код, а также не изобретайте сложные схемы, чтобы избежать дублирования трех строк кода.

Мы будем производить двумерные срезы КТ с несколькими каналами. В дополнительных каналах будут храниться соседние срезы КТ. Вспомните рис. 4.2, который мы повторим здесь как рис. 13.12. На нем видно, что каждый срез компьютерной томографии можно рассматривать как двумерное изображение в градациях серого.

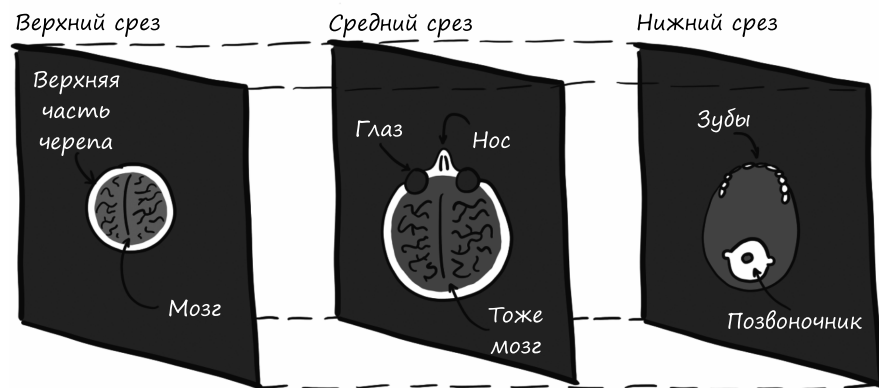


Рис. 13.12. Каждый срез КТ — это положение в пространстве

Способ объединения этих фрагментов зависит от нас. Для ввода в нашу модель классификации мы рассматривали эти срезы как трехмерный массив данных и использовали трехмерные свертки для обработки каждого элемента. Вместо этого в модели сегментации мы будем рассматривать каждый срез как один канал и создавать многоканальное 2D-изображение. Это будет означать, что мы рассматриваем каждый срез компьютерной томографии, как если бы это был цветовой канал изображения RGB, как мы видели на рис. 4.1 и как показано на рис. 13.13. Входные срезы КТ будут сложены вместе и использоваться так же, как и любое другое 2D-изображение. Каналы совмещенного КТ-изображения не

будут соответствовать цветам, но в 2D-свертках и не требуется, чтобы входные каналы были именно цветами, поэтому все будет работать нормально.

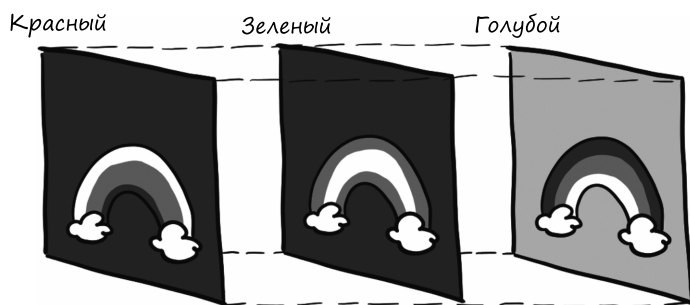


Рис. 13.13. Каждый канал фотографического изображения представляет свой цвет

Для проверки нам потребуется создать по одному элементу данных на срез КТ, у которого есть запись в положительной маске, для каждого имеющегося у нас проверочного КТ. Поскольку разные КТ-сканы могут иметь разное количество срезов¹, мы собираемся ввести новую функцию, которая кэширует размер каждого КТ-снимка и его положительную маску на диск. Нам это нужно, чтобы иметь возможность быстро создать полный проверочный набор, не загружая каждую КТ при инициализации набора данных. Мы продолжим использовать тот же кэширующий декоратор, что и раньше. Заполнение этих данных также будет происходить во время сценария `prepcache.py`, который мы должны запустить один раз, прежде чем начинать обучение модели (листинг 13.11).

Листинг 13.11. `dsets.py:220`

```
@raw_cache.memoize(typed=True)
def getCtSampleSize(series_uid):
    ct = Ct(series_uid)
    return int(ct.hu_a.shape[0]), ct.positive_indexes
```

Большая часть метода `Luna2dSegmentationDataset.__init__` подобна тому, что мы видели раньше. У нас есть новый параметр `contextSlices_count`, а также `augmentation_dict`, аналогичный тому, что мы представили в главе 12.

Обработку флага, указывающего, предназначен ли данный набор для обучения или проверки, нужно несколько изменить. Поскольку мы больше не выполняем обучение на отдельных узелках, нам придется разделить список серий на обучающий и проверочный наборы. Это означает, что весь КТ-снимок вместе

¹ Большинство КТ-сканеров производят срезы размером 512×512 , и мы не будем беспокоиться о тех, которые делают что-то другое.

со всеми содержащимися в нем кандидатами в узелки будет либо в обучающем наборе, либо в проверочном (листинг 13.12).

Листинг 13.12. `dssets.py:242, __init__`

```
if isValSet_bool:
    assert val_stride > 0, val_stride
    self.series_list = self.series_list[:,val_stride]
    assert self.series_list
elif val_stride > 0:
    del self.series_list[:,val_stride]
    assert self.series_list
```

Из списка, содержащего все серии, мы сохраняем только каждый `val_stride`-й элемент, начиная с 0

Если выполняется обучение, вместо этого мы удаляем каждый `val_stride`-й элемент

У нас будет два режима, в которых мы будем проверять качество обучения. Для начала, когда `fullCt_bool` имеет значение `True`, для данного набора мы будем использовать каждый срез в КТ. Это поможет оценить сквозную производительность, поскольку нам нужно сделать вид, что никакой предварительной информации о КТ у нас нет. Второй режим служит для проверки во время обучения, когда мы ограничиваем себя только срезами КТ, содержащими положительную маску.

Поскольку теперь мы хотим, чтобы учитывались только определенные серии КТ, мы перебираем UID серий, которые нам нужны, и получаем общее количество срезов и список интересных срезов (листинг 13.13).

Листинг 13.13. `dssets.py:250, __init__`

```
self.sample_list = []
for series_uid in self.series_list:
    index_count, positive_indexes = getCtSampleSize(series_uid)

    if self.fullCt_bool:
        self.sample_list += [(series_uid, slice_ndx)
                             for slice_ndx in range(index_count)]
    else:
        self.sample_list += [(series_uid, slice_ndx)
                             for slice_ndx in positive_indexes]
```

Здесь мы расширяем `sample_list` срезами КТ, используя диапазон...

...а здесь берем только интересные срезы

Таким образом, наша проверка будет относительно быстрой и гарантирует, что мы получим полную статистику истинно положительных и ложноотрицательных результатов, и мы делаем предположение, что другие срезы будут содержать ложноположительные и истинно отрицательные статистические данные, относительно похожие на те, которые мы оценивали во время проверки.

Имея набор значений `series_uid`, мы можем отфильтровать список `candidateInfo_list`, чтобы он содержал только узелки-кандидаты с `series_uid`, включенным в этот набор. Кроме того, мы создадим еще один список, содержащий только положительные кандидаты, чтобы во время обучения мы могли использовать их в качестве обучающих (листинг 13.14).

Листинг 13.14. `dsets.py:261, __init__`

```

self.candidateInfo_list = getCandidateInfoList()  ← Это кэшируется

series_set = set(self.series_list)  ← Подготовка к более быстрому поиску
self.candidateInfo_list = [cit for cit in self.candidateInfo_list
                           if cit.series_uid in series_set]  ← Отбор кандидатов, ID которых
                                                             не присутствует в наборе

self.pos_list = [nt for nt in self.candidateInfo_list
                 if nt.isNodule_bool]  ← Так как впереди будет балансировка данных,
                                       нам нужен список настоящих узелков

```

Наша реализация `__getitem__` также будет чуть более изящной, поскольку делегирует большую часть логики функции, которая упрощает получение определенного образца. По сути, мы хотели бы получить данные в трех разных формах. Во-первых, у нас есть полный срез КТ, как указано в параметрах `series_uid` и `ct_ndx`. Во-вторых, у нас есть область вокруг узелка, которую мы будем применять для обучающих данных (мы объясним, почему не используем полные срезы). Наконец, `DataLoader` будет запрашивать образцы через целое число `ndx`, и набор данных должен будет вернуть тип — обучающий или проверочный.

Функции `__getitem__` базового класса или подкласса преобразуют целое число `ndx` либо в полный срез, либо в обучающий фрагмент, в зависимости от ситуации. Как уже упоминалось, `__getitem__` проверочного набора просто вызывает другую функцию, которая делает всю работу. Перед этим индекс помещается в список элементов данных, чтобы отделить размер эпохи (определяемый длиной набора данных) от фактического количества элементов (листинг 13.15).

Листинг 13.15. `dsets.py:281, __getitem__`

```

def __getitem__(self, ndx):  ← Операция взятия остатка реализует перенос
    series_uid, slice_ndx = self.sample_list[ndx % len(self.sample_list)]
    return self.getitem_fullSlice(series_uid, slice_ndx)

```

Это было легко, но нам еще нужно реализовать интересную функциональность метода `getitem_fullSlice` (листинг 13.16).

Листинг 13.16. `dsets.py:285, getitem_fullSlice`

```

def getitem_fullSlice(self, series_uid, slice_ndx):
    ct = getCt(series_uid)
    ct_t = torch.zeros((self.contextSlices_count * 2 + 1, 512, 512))

    start_ndx = slice_ndx - self.contextSlices_count
    end_ndx = slice_ndx + self.contextSlices_count + 1
    for i, context_ndx in enumerate(range(start_ndx, end_ndx)):
        context_ndx = max(context_ndx, 0)
        context_ndx = min(context_ndx, ct.hu_a.shape[0] - 1)
        ct_t[i] = torch.from_numpy(ct.hu_a[context_ndx].astype(np.float32))
    ct_t.clamp_(-1000, 1000)

```

Предварительное распределение вывода

Когда мы выходим за пределы `ct_a`, мы дублируем первый или последний срез

```
pos_t = torch.from_numpy(ct.positive_mask[slice_ndx]).unsqueeze(0)

return ct_t, pos_t, ct.series_uid, slice_ndx
```

Подобное разделение функций означает, что мы всегда можем запросить набор данных для определенного среза (или обрезанного обучающего фрагмента, который мы увидим в следующем подразделе) по UID и положению. Для целочисленного индексирования мы выполняем метод `__getitem__`, впоследствии получающий элемент данных из перетасованного списка.

Помимо `ct_t` и `pos_t`, остальная часть возвращаемого кортежа состоит из информации для отладки и отображения. Для обучения нам это не нужно.

13.5.5. Разработка наших данных для обучения и проверки

Прежде чем мы приступим к реализации обучающего набора данных, следует пояснить, почему обучающие данные будут отличаться от проверочных. Вместо полных срезов КТ мы будем выполнять обучение на фрагментах 64×64 вокруг положительных кандидатов (настоящих узелков). Эти фрагменты 64×64 будут взяты случайным образом из области 96×96 в центре узелка. В качестве дополнительных «каналов» для 2D-сегментации мы также включим по три фрагмента сверху и снизу.

Этот подход позволяет сделать обучение более стабильным и ускорить сходимость. Причина в том, что мы пытались выполнять обучение на целых срезах КТ, но результаты оказались неудовлетворительными. После некоторых экспериментов мы обнаружили, что метод полуслучайной вырезки фрагментов 64×64 работает достаточно хорошо, поэтому решили использовать его для книги. Работая над собственными проектами, вы должны будете проводить такие же эксперименты!

Мы считаем, что обучение с помощью всего фрагмента оказалось нестабильным из-за проблемы с балансировкой классов. Поскольку каждый узелок весьма мал по сравнению со всем срезом КТ, мы снова оказались в ситуации «иголка в стоге сена», похожей на ту, из которой мы с трудом выбрались в предыдущей главе, когда положительные элементы данных были завалены отрицательными. В данном случае мы говорим о пикселях, а не об узелках, но суть та же. Выполняя обучение на фрагментах, мы сохраняем то же количество положительных пикселей, но уменьшаем количество отрицательных пикселей на несколько порядков.

Поскольку наша модель сегментации является попиксельной и принимает изображения произвольного размера, мы можем обойтись без обучения и проверки

на выборках с разными размерами. При проверке используются те же свертки с теми же весами, но все это применяется к большему набору пикселей (и, следовательно, меньше граничных пикселей заполняется краевыми данными).

Для этого подхода важно отметить, что, поскольку проверочный набор содержит на несколько порядков больше отрицательных пикселей, во время проверки модель будет выдавать огромный процент ложноположительных результатов. Существует множество способов обмануть модель сегментации! Не помогает и то, что мы хотим сохранить высокий уровень отклика. Мы обсудим это подробнее в подразделе 13.6.3.

13.5.6. Реализация набора данных `TrainingLuna2dSegmentation`

С этим разобрались, теперь вернемся к коду. Ниже приведен метод `__getitem__` обучающего набора. Он выглядит точно так же, как и для проверочного набора, за исключением того, что теперь мы делаем выборку из `pos_list` и вызываем метод `getItem_trainingCrop` с кортежем информации о кандидате, поскольку нам нужны серия и точное расположение центра, а не только срез (листинг 13.17).

Листинг 13.17. `dsets.py:320, __getitem__`

```
def __getitem__(self, ndx):
    candidateInfo_tup = self.pos_list[ndx % len(self.pos_list)]
    return self.getItem_trainingCrop(candidateInfo_tup)
```

Для реализации `getItem_trainingCrop` мы задействуем функцию `getCtRawCandidate`, аналогичную той, которую использовали при обучении классификации. Здесь мы передаем фрагмент другого размера, но сама функция не изменилась, за исключением того, что теперь она возвращает дополнительный массив с фрагментом `ct.positive_mask`.

Мы ограничиваем значение `pos_a` центральным срезом, который на самом деле сегментируем, а затем строим наши случайные кадры 64×64 из 96×96 , которые мы получили от `getCtRawCandidate`. Получив их, мы возвращаем кортеж с теми же элементами, что и в проверочном наборе (листинг 13.18).

Листинг 13.18. `dsets.py:324, getItem_trainingCrop`

```
def getItem_trainingCrop(self, candidateInfo_tup):
    ct_a, pos_a, center_irc = getCtRawCandidate(
        candidateInfo_tup.series_uid,
        candidateInfo_tup.center_xyz,
        (7, 96, 96),
    )
    pos_a = pos_a[3:4]
```

← Получаем кандидат и небольшой фрагмент вокруг него

← Срез из одного элемента сохраняет третье измерение, которое будет (единственным) выходным каналом

```

row_offset = random.randrange(0, 32)
col_offset = random.randrange(0, 32)
ct_t = torch.from_numpy(ct_a[:, row_offset:row_offset+64,
                             col_offset:col_offset+64]).to(torch.float32)
pos_t = torch.from_numpy(pos_a[:, row_offset:row_offset+64,
                             col_offset:col_offset+64]).to(torch.long)

slice_ndx = center_irc.index

return ct_t, pos_t, candidateInfo_tup.series_uid, slice_ndx

```

Получаем два случайных числа от 0 до 31 и обрезаем КТ и маску

Вы могли заметить, что в этой реализации набора данных не создано дополнение. На сей раз мы реализуем его немного по-другому: выполним дополнение данных на ГП.

13.5.7. Дополнение данных на ГП

Одна из ключевых проблем, которые возникают при обучении модели глубокого обучения, — необходимость устранять узкие места в конвейере обучения. Правда в том, что узкое место будет всегда¹. Хитрость заключается в том, чтобы убедиться, что узкое место находится в самом дорогом и трудном для обновления ресурсе и использование этого ресурса не является расточительным.

Часто узкие места возникают в следующих процессах:

- в конвейере загрузки данных во время работы с необработанным вводом-выводом или при распаковке данных, когда они находятся в ОЗУ. Мы решили эту проблему с помощью библиотеки `diskcache`;
- при предварительной обработке загружаемых данных в процессоре. Обычно это нормализация или дополнение данных;
- в цикле обучения на ГП. Обычно нам лучше иметь узкое место именно здесь, поскольку общие затраты на операции глубокого обучения у графических процессоров обычно выше, чем у хранилища или ЦП;
- реже узким местом становится *пропускная способность памяти* между ЦП и ГП. Это означает, что ГП выполняет не так уж много работы по сравнению с объемом перемещаемых данных.

Поскольку графические процессоры при выполнении определенного класса задач могут быть в 50 раз быстрее, чем центральные, часто имеет смысл переносить эти задачи на ГП с ЦП, если последний слишком сильно нагружается. Это особенно верно в случаях, когда данные дополняются во время этой обработки, так как

¹ Иначе модель обучалась бы мгновенно!

при перемещении меньшего объема входных данных в графический процессор дополненные данные остаются для него локальными, поэтому приходится перемещать меньше данных.

В нашем случае мы собираемся перенести дополнение данных на ГП. Это снизит нагрузку на ЦП, а ГП легко справится с дополнительной нагрузкой. Лучше нагрузить ГП небольшой дополнительной работой, чем наблюдать его простой, пока процессор будет дополнять данные.

Мы реализуем это с помощью второй модели, похожей на все другие подклассы `nn.Module`, которые встречали в книге ранее. Основное различие состоит в том, что нас не интересует обратное распространение градиентов по модели, а метод `forward` будет наделен совершенно другим функционалом. Кроме того, мы внесем небольшие изменения в сами процедуры дополнения, поскольку в данной главе мы работаем с 2D-данными, но в остальном дополнение будет очень похоже на то, что мы видели в главе 12. Модель будет брать на вход тензоры и создавать другие тензоры, как и другие модели, которые мы реализовали.

Конструктор модели принимает те же аргументы, необходимые для дополнения данных, — `flip`, `offset` и т. д., которые мы использовали в предыдущей главе, и присваивает их `self` (листинг 13.19).

Листинг 13.19. `model.py:56`, класс `SegmentationAugmentation`

```
class SegmentationAugmentation(nn.Module):
    def __init__(
        self, flip=None, offset=None, scale=None, rotate=None, noise=None
    ):
        super().__init__()

        self.flip = flip
        self.offset = offset
        # ... строка 64
```

Метод дополнения `forward` принимает входные данные и метку и вызывает создание тензора `transform_t`, который затем будет управлять нашими вызовами `affine_grid` и `grid_sample`. Эти вызовы после главы 12 должны показаться вам знакомыми (листинг 13.20).

Теперь, когда ясно, что нужно сделать с `transform_t`, чтобы получить данные, взглянем на функцию `_build2dTransformMatrix`, которая фактически создает нужную нам матрицу преобразования (листинг 13.21).

Если не считать небольших различий, связанных с 2D-данными, код дополнения данных для графического процессора очень похож на тот же код для ЦП. Это здорово, поскольку мы получаем код, для которого не имеет значения, где он выполняется. Основное различие заключается не в базовой реализации, а в том, как мы поместили эту реализацию в подкласс `nn.Module`. Мы рассматривали

модели исключительно как инструмент глубокого обучения, но теперь стало ясно, что тензоры PyTorch можно использовать несколько шире. Имейте это в виду, когда начнете работу над следующим проектом. Разнообразие задач, которые вы можете выполнить с помощью тензора с ускорением на ГП, весьма велико!

Листинг 13.20. model.py:68, SegmentationAugmentation.forward

```
def forward(self, input_g, label_g):
    transform_t = self._build2dTransformMatrix()
    transform_t = transform_t.expand(input_g.shape[0], -1, -1)
    transform_t = transform_t.to(input_g.device, torch.float32)
    affine_t = F.affine_grid(transform_t[:, :2],
                             input_g.size(), align_corners=False)

    augmented_input_g = F.grid_sample(input_g,
                                      affine_t, padding_mode='border',
                                      align_corners=False)
    augmented_label_g = F.grid_sample(label_g.to(torch.float32),
                                      affine_t, padding_mode='border',
                                      align_corners=False)

    if self.noise:
        noise_t = torch.randn_like(augmented_input_g)
        noise_t *= self.noise

        augmented_input_g += noise_t

    return augmented_input_g, augmented_label_g > 0.5
```

Обратите внимание, что мы дополняем 2D-данные

Первое измерение данного преобразования — это пакет, но нам нужны только первые две строки матриц 3×3 для каждого элемента пакета

Нам нужно, чтобы одно и то же преобразование применялось к КТ и маске, поэтому мы используем ту же сетку. Поскольку `grid_sample` работает только с числами с плавающей запятой, выполняется преобразование

Непосредственно перед возвратом мы преобразуем маску обратно в булевы значения, сравнивая их с 0,5. Интерполяция `grid_sample` дает дробные значения

Листинг 13.21. model.py:90, _build2dTransformMatrix

```
def _build2dTransformMatrix(self):
    transform_t = torch.eye(3)

    for i in range(2):
        if self.flip:
            if random.random() > 0.5:
                transform_t[i, i] *= -1
        # ... строка 108
        if self.rotate:
            angle_rad = random.random() * math.pi * 2
            s = math.sin(angle_rad)
            c = math.cos(angle_rad)

            rotation_t = torch.tensor([
                [c, -s, 0],
                [s, c, 0],
                [0, 0, 1]])

            transform_t @= rotation_t

    return transform_t
```

Создание матрицы 3×3 , у которой мы позже удалим последнюю строку

Опять же, здесь мы дополняем 2D-данные

Принимает случайный угол в радианах, то есть в диапазоне $0 \dots 2\{\pi\}$

Матрица вращения для двумерного поворота на случайный угол в первых двух измерениях

Применение вращения к матрице преобразования с помощью оператора умножения матриц Python

13.6. ВНЕДРЕНИЕ СЕГМЕНТАЦИИ В СЦЕНАРИЙ ОБУЧЕНИЯ

У нас есть модель и данные. Пора их использовать, и этап 2В на рис. 13.14 так и говорит: нужно обучить новую модель на новых данных.

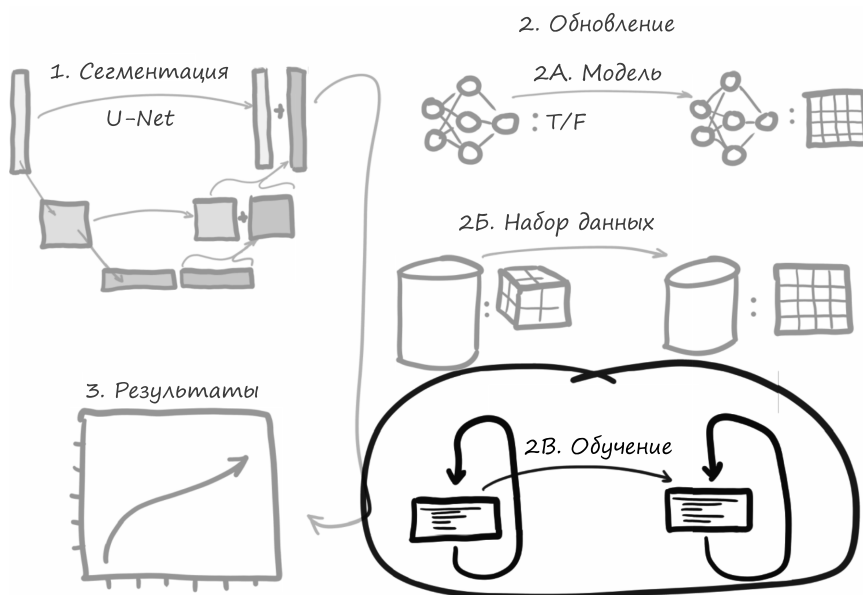


Рис. 13.14. План этого раздела — изменения в цикле обучения

Для достижения большей точности в процессе обучения модели мы обновим три вещи, влияющие на результат обучения, который получили в главе 12:

- нам нужно создать экземпляр новой модели (что неудивительно);
- мы введем новое значение потери — потери Дайса;
- мы также рассмотрим оптимизатор, отличный от SGD, который мы использовали до сих пор. Это тоже будет популярный вариант, а именно оптимизатор Adam.

Не забудем и про анализ результатов, добавив:

- логирование изображений для визуального контроля сегментации в TensorBoard;
- логирование новых метрик в TensorBoard;
- сохранение лучшей модели по результатам проверки.

В целом обучающий сценарий `p2ch13/training.py` будет больше похож на тот, который мы использовали для обучения классификации в главе 12, чем на адаптированный код из этой главы. Любые существенные изменения будут описаны, но имейте в виду, что отдельные незначительные изменения будут опущены. Чтобы просмотреть весь код, скачайте его из [GitHub](#).

13.6.1. Инициализация наших моделей сегментации и увеличения

Метод `initModel` довольно тривиален. Мы используем класс `UNetWrapper` и зададим ему параметры конфигурации, которые вскоре рассмотрим подробно. Кроме того, теперь у нас есть вторая модель для дополнения. Как и раньше, при желании мы можем переместить эту модель на ГП и, возможно, настроить обучение на нескольких ГП с помощью `DataParallel`. Но здесь мы опустим эти административные задачи (листинг 13.22).

Листинг 13.22. `training.py:133, .initModel`

```
def initModel(self):
    segmentation_model = UNetWrapper(
        in_channels=7,
        n_classes=1,
        depth=3,
        wf=4,
        padding=True,
        batch_norm=True,
        up_mode='upconv',
    )

    augmentation_model = SegmentationAugmentation(**self.augmentation_dict)

    # ... строка 154
    return segmentation_model, augmentation_model
```

Для ввода в `UNet` у нас есть семь каналов: один срез, который в данный момент сегментируется, и 3 + 3 среза контекста вокруг него. У нас будет один выходной класс, указывающий, является ли данный воксель частью узелка. Параметр глубины определяет, насколько глубока эта воображаемая буква U. Каждая операция понижающей дискретизации добавляет 1 к ее глубине. Параметр `wf=5` означает, что в первом слое будет $2 * wf == 32$ фильтра, и это число удваивается при каждом понижении частоты дискретизации. Нам нужно будет дополнить свертки, чтобы получить выходное изображение того же размера, что и входное. Мы также хотим выполнить пакетную нормализацию внутри сети после каждой функции активации, и наша функция повышения дискретизации должна быть слоем свертки с повышением частоты, реализованным `nn.ConvTranspose2d` (см. `util/unet.py`, строка 123).

13.6.2. Использование оптимизатора Adam

Оптимизатор Adam (или просто Adam) (<https://arxiv.org/abs/1412.6980>) — это одна из альтернатив SGD для обучения моделей. Он поддерживает выбор отдельной скорости обучения для каждого параметра и автоматически изменяет ее по мере прохождения обучения. Благодаря этой автоматической подстройке нам обычно не приходится задавать какую-то нестандартную скорость обучения, и оптимизатор быстро определит разумную скорость обучения сам.

Создадим экземпляр Adam в коде (листинг 13.23).

Листинг 13.23. training.py:156, .initOptimizer

```
def initOptimizer(self):  
    return Adam(self.segmentation_model.parameters())
```

Для большинства проектов на начальном этапе Adam считается хорошим оптимизатором¹. Обычно можно найти конфигурацию стохастического градиентного спуска с импульсом Нестерова, которая будет работать лучше Adam, но подобрать правильные гиперпараметры для инициализации SGD в данном проекте может быть сложно и на это уйдет много времени.

Существует множество вариаций Adam — AdaMax, RAdam, Ranger и т. д. У каждой из них есть свои сильные и слабые стороны. Подробности их работы выходят за рамки темы этой книги, но мы полагаем, что вам стоит хотя бы знать о существовании таких альтернатив в принципе.

13.6.3. Потеря Дайса

Коэффициент Сёренсена — Дайса (https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient), также известный как *потеря Дайса*, служит в качестве метрики потерь в задачах сегментации. Одним из преимуществ использования потери Дайса, по сравнению с перекрестно-энтропийной потерей на пиксель, является то, что потеря Дайса обрабатывает случай, когда лишь небольшая часть изображения помечена положительной. Как мы помним из раздела 11.10, при использовании кросс-энтропийных потерь несбалансированные обучающие данные создают проблемы. Здесь у нас именно такая ситуация — большая часть КТ не является узелком. К счастью, при использовании потери Дайса это не проблема.

Коэффициент Сёренсена — Дайса — это отношение правильно сегментированных пикселей к сумме предсказанных и фактических пикселей. Эти отношения показаны на рис. 13.15. Слева приведена иллюстрация метрики Дайса. Это удвоенная площадь пересечения (*истинно положительные результаты*, заштрихованная область), деленная на сумму всей прогнозируемой площади и всей области,

¹ См. <http://cs231n.github.io/neural-networks-3>.

помеченной как истинная (пересечение считается дважды). Справа показаны два примера высокого и низкого коэффициента Дайса.

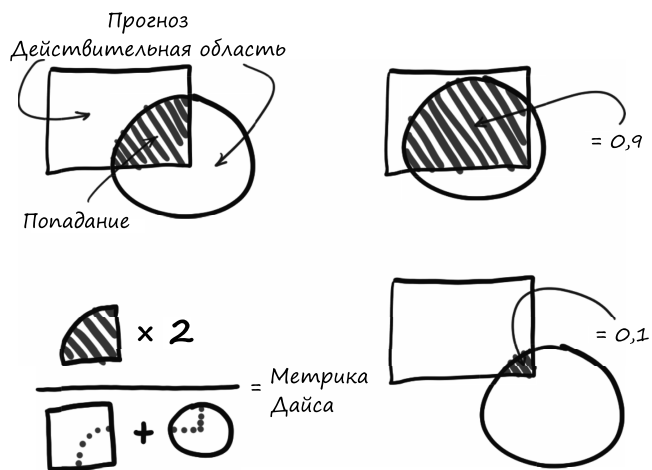


Рис. 13.15. Примеры высокого и низкого коэффициента Дайса

Звучит знакомо. На самом деле именно это соотношение мы видели в главе 12. По сути, мы собираемся для каждого пикселя использовать метрику F1!

ПРИМЕЧАНИЕ

Мы получили метрику F1 для каждого пикселя, где под популяцией имеются в виду пиксели одного изображения. Поскольку популяция полностью содержится в одной обучающей выборке, мы можем использовать ее непосредственно для обучения. В случае классификации метрику F1 нельзя вычислить для одной мини-партии, и, следовательно, мы не можем применить ее непосредственно для обучения.

Поскольку `label_g` — это фактически логическая маска, мы можем умножить ее на прогнозы, чтобы получить истинно положительные результаты. Обратите внимание: `prediction_devtensor` не рассматривается как логическое значение. Определенная на нем потеря не будет дифференцируемой. Вместо этого мы заменяем количество истинных срабатываний на сумму предсказанных значений для пикселей, где истинное значение равно 1. В результате значения прогнозов будут приближаться к 1, но иногда будут получаться неопределенные прогнозы в диапазоне от 0,4 до 0,6. Эти неопределенные значения будут вносить примерно одинаковый вклад в обновления градиента, независимо от того, на какую сторону от 0,5 они попадают. Коэффициент Дайса, использующий непрерывные прогнозы, иногда называют *мягким коэффициентом Дайса*.

Есть еще одно крошечное осложнение. Поскольку мы хотим, чтобы потери были минимальными, мы возьмем полученное отношение и вычтем его из 1. Это

инвертирует наклон функции потерь, так что в случае большого перекрытия наши потери будут низкими и при низком перекрытии — высокими. Вот как это выглядит в коде (листинг 13.24).

Листинг 13.24. training.py:315, .diceLoss

Суммируем все, кроме измерения пакета, чтобы получить положительно помеченные, (мягко) положительно обнаруженные и (мягко) истинно положительные результаты для каждого элемента пакета

```
def diceLoss(self, prediction_g, label_g, epsilon=1):
    → diceLabel_g = label_g.sum(dim=[1,2,3])
    dicePrediction_g = prediction_g.sum(dim=[1,2,3])
    diceCorrect_g = (prediction_g * label_g).sum(dim=[1,2,3])

    diceRatio_g = (2 * diceCorrect_g + epsilon) \
        / (dicePrediction_g + diceLabel_g + epsilon)
```

Соотношение Дайса.
Чтобы избежать случаев, когда у нас нет ни прогнозов, ни меток, мы добавляем 1 и к числителю, и к знаменателю

```
return 1 - diceRatio_g
```

Чтобы получить потери, вычитаем отношение из единицы, чтобы потери стремились к минимуму

Мы собираемся обновить функцию `calculateBatchLoss`, чтобы она вызывала `self.diceLoss`. Дважды. Мы вычислим нормальную потерю Дайса для обучающей выборки, а также для пикселей, включенных в `label_g`. Умножая наши прогнозы (которые, как вы помните, представляют собой значения с плавающей запятой) на метки (по сути, являющиеся логическими значениями), мы получаем псевдопрогнозы, в которых каждый отрицательный пиксель будет «истинно правильным» (поскольку все значения для этих пикселей умножаются на значения `false-is-zero` из `label_g`). Единственные пиксели, которые будут генерировать потери, — ложноотрицательные (все, что должно было быть предсказано как истинное, но не предсказалось). Это будет полезно, поскольку нам важно получить высокий отклик. В конце концов, мы не можем правильно классифицировать опухоли, если не обнаружим их (листинг 13.25)!

Листинг 13.25. training.py:282, .computeBatchLoss

```
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g,
    classificationThreshold=0.5):
```

```
    input_t, label_t, series_list, _slice_ndx_list = batch_tup
```

```
    input_g = input_t.to(self.device, non_blocking=True)
    label_g = label_t.to(self.device, non_blocking=True)
```

Передача в ГП

```
    if self.segmentation_model.training and self.augmentation_dict:
        input_g, label_g = self.augmentation_model(input_g, label_g)
```

Дополнение, нужное при обучении. На этапе проверки данный шаг опускается

```
    prediction_g = self.segmentation_model(input_g)
```

Запуск модели сегментации...

```
    diceLoss_g = self.diceLoss(prediction_g, label_g)
    fnLoss_g = self.diceLoss(prediction_g * label_g, label_g)
```

...и применение потери Дайса

```
    # ... строка 313
```

```
    return diceLoss_g.mean() + fnLoss_g.mean() * 8
```

Так-так. А это что?

Немного поговорим о том, что мы делаем с оператором возврата `diceLoss_g.mean() + fnLoss_g.mean() * 8`.

Взвешивание потерь

В главе 12 мы обсуждали формирование набора данных, чтобы наши классы не были слишком уж несбалансированными. Это помогло получить сходящийся процесс обучения, поскольку положительные и отрицательные элементы, присутствующие в каждом пакете, могли противодействовать общему притяжению друг друга, и модель должна была научиться различать их, чтобы совершенствоваться. Здесь мы достигаем такого же баланса, обрезая наши обучающие фрагменты, чтобы включить меньше неположительных пикселей. Кроме того, невероятно важно иметь высокий отклик, и мы должны убедиться, что во время обучения потери стремятся именно к этому.

У нас будет *взвешенная потеря*, в которой у одного класса есть приоритет по сравнению с другим. Умножая `fnLoss_g` на 8, мы говорим, что правильное определение всей совокупности положительных пикселей в восемь раз важнее, чем правильное получение всей совокупности отрицательных пикселей (в девять, если считать единицу в `diceLoss_g`). Поскольку область, охватываемая положительной маской, намного меньше, чем весь кадр 64×64 , это также означает, что каждый отдельный положительный пиксель должен сильнее влиять на коэффициенты во время обратного распространения.

Мы готовы обменять какое-то количество правильно предсказанных отрицательных пикселей в общей потере Дайса на один правильный пиксель в ложноотрицательной потере. Поскольку общие потери являются строгим надмножеством ложноотрицательных потерь, для подобного обмена можно использовать лишь истинно отрицательные пиксели (так как все истинно положительные пиксели уже включены в ложноотрицательные потери).

Мы готовы пожертвовать огромным количеством истинных отрицательных пикселей в погоне за лучшим откликом, поэтому в целом стоит ожидать большого количества ложных срабатываний¹. Все дело в том, что в нашей задаче очень-очень важен хороший отклик, и мы бы предпочли получить несколько ложноположительных результатов, чем хотя бы один ложноотрицательный.

Следует отметить: этот подход работает только в случае применения оптимизатора Adam. При использовании SGD толчок к завышению прогноза приведет к тому, что каждый пиксель начнет определяться как положительный. Способность оптимизатора Adam точно настраивать скорость обучения означает, что подчеркивание ложноотрицательных потерь можно преодолеть.

¹ Роксы бы гордилась!

Вычисление метрик

Мы собираемся намеренно исказить наши числа для получения лучшего отклика. Посмотрим, что из этого выйдет. В классификации `calculateBatchLoss` для каждого элемента данных мы вычисляем различные значения, которые использовали для метрик и т. п. Мы также вычисляем аналогичные значения для общих результатов сегментации. Эти истинно положительные и другие метрики ранее были рассчитаны в `logMetrics`, но из-за размера данных результатов (напомним, что каждый отдельный срез КТ из набора проверки составляет четверть миллиона пикселей!) нам необходимо вычислить сводную статистику в функции `calculateBatchLoss` (листинг 13.26).

Листинг 13.26. `training.py:297, .computeBatchLoss`

```
start_ndx = batch_ndx * batch_size
end_ndx = start_ndx + input_t.size(0)

with torch.no_grad():
    predictionBool_g = (prediction_g[:, 0:1]
                        > classificationThreshold).to(torch.float32)

    tp = ( predictionBool_g * label_g).sum(dim=[1,2,3])
    fn = ((1 - predictionBool_g) * label_g).sum(dim=[1,2,3])
    fp = ( predictionBool_g * (~label_g)).sum(dim=[1,2,3])

    metrics_g[METRICS_LOSS_NDX, start_ndx:end_ndx] = diceLoss_g
    metrics_g[METRICS_TP_NDX, start_ndx:end_ndx] = tp
    metrics_g[METRICS_FN_NDX, start_ndx:end_ndx] = fn
    metrics_g[METRICS_FP_NDX, start_ndx:end_ndx] = fp
```

Мы ограничиваем прогноз, чтобы получить «жесткую» метрику Дайса, но конвертируем его в число с плавающей запятой для последующего умножения

Вычисление истинно положительных, ложноположительных и ложноотрицательных результатов аналогично тому, что мы делали при вычислении потери Дайса

Метрики хранятся в большом тензоре для дальнейшего использования. Метрика вычисляется по элементам пакета, а не усредняется по пакету

Как мы обсуждали в начале подраздела, можно вычислить истинно положительные и другие результаты, умножив прогноз (или его отрицание) на метку (или ее отрицание). Поскольку точные значения наших прогнозов нас не слишком волнуют (не имеет значения, равна вероятность пикселя 0,6 или 0,9, так как для рассмотрения его как кандидата достаточно просто превысить пороговое значение), мы собираемся создать `predictionBool_g`, сравнив его с пороговым значением 0,5.

13.6.4. Получение изображений в TensorBoard

Одна из приятных особенностей задач сегментации заключается в том, что результат работы легко представить визуально. А возможность визуальной оценки результата позволяет определить, развивается ли модель (возможно, ей нужно еще чуть-чуть обучиться) или вообще делает не то (и тогда не стоит тратить время на дальнейшее обучение). Существует много способов представить результаты в виде изображений и много способов их отобразить. TensorBoard

отлично поддерживает работу с такими данными, и у нас уже есть экземпляры `TensorBoard SummaryWriter`, которые используются в цикле обучения, поэтому мы собираемся применить именно `TensorBoard`. Посмотрим, что нужно сделать, чтобы все заработало.

Мы добавим функцию `logImages` в основной класс приложения и вызовем ее вместе с загрузчиками данных для обучения и проверки. И, поскольку мы здесь, изменим кое-что в обучающем цикле. Мы собираемся выполнять проверку и логирование изображений только в первую, а затем в каждую пятую эпоху. Для этого будем сверять номер эпохи с новой константой `validation_cadence`.

Во время обучения нужно будет сбалансировать несколько вещей:

- приблизительно представлять, как идет обучение модели, без необходимости ждать слишком долго;
- выделять большую часть наших циклов ГП на обучение, а не на проверку;
- проверять качество обучения на проверочном наборе.

Первый пункт означает следующее: эпохи должны быть относительно короткими, чтобы можно было чаще вызывать `logMetrics`. Второй — что перед вызовом `doValidation` должно успеть пройти какое-то обучение. Третий пункт означает, что нам нужно вызывать `doValidation` регулярно, а не один раз в конце обучения. Выполняя проверку только в первую, а затем в каждую пятую эпоху, мы можем добиться всех этих целей. Мы будем понимать, как идет прогресс обучения, выделять большую часть нашего времени на обучение и периодически выполнять проверки (листинг 13.27).

Листинг 13.27. `training.py:210, SegmentationTrainingApp.main`

```
def main(self):
    # ... строка 217
    self.validation_cadence = 5
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        # ... строка 228
        trnMetrics_t = self.doTraining(epoch_ndx, train_dl)
        self.logMetrics(epoch_ndx, 'trn', trnMetrics_t)

        if epoch_ndx == 1 or epoch_ndx % self.validation_cadence == 0:
            # ... строка 239
            self.logImages(epoch_ndx, 'trn', train_dl)
            self.logImages(epoch_ndx, 'val', val_dl)
```

Внешний цикл прохода по эпохам

Логирование метрик после каждой эпохи

Проверка модели через определенный интервал...

Обучение одной эпохи

...с логированием изображений

Не существует единственно верного способа структурировать логирование изображений. Мы возьмем несколько КТ как из обучающего, так и из проверочного

набора. Для каждой КТ мы выберем шесть равномерно распределенных срезов и сравним реальные данные и выходные данные нашей модели. Мы выбрали шесть срезов только потому, что TensorBoard будет выводить по 12 изображений за раз, и можем расположить окно браузера так, чтобы поверх выходных данных модели отображался ряд изображений меток. Подобное расположение элементов позволяет легко визуальное сравнить данные, как мы можем видеть на рис. 13.16.

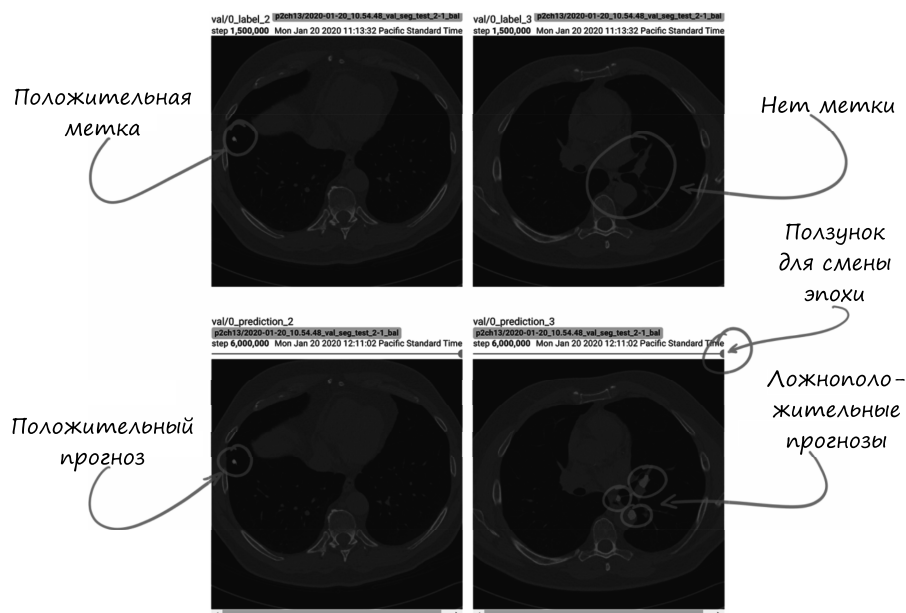


Рис. 13.16. Верхний ряд: метки обучающих данных.
Нижний ряд: результат сегментации

Обратите также внимание на маленькую точку-ползунок на изображениях prediction. Он позволит нам просматривать предыдущие версии изображений с той же меткой (например, val/0_prediction_3, но в более раннюю эпоху). Возможность видеть, как выходные данные сегментации меняются с течением времени, может быть полезна, когда мы пытаемся что-то отладить или внести изменения для достижения определенного результата. По мере обучения TensorBoard будет ограничивать количество изображений в этой истории до десяти, чтобы не перегружать браузер.

Код, который создает этот вывод, начинается с получения 12 серий из соответствующего загрузчика данных и шести изображений из каждой серии (листинг 13.28).

Листинг 13.28. training.py:326, .logImages

```
def logImages(self, epoch_ndx, mode_str, dl):
    self.segmentation_model.eval()  # ← Задание модели для оценки

    images = sorted(dl.dataset.series_list)[:12]  # ← Берем те же 12 КТ в обход загрузчика
    for series_ndx, series_uid in enumerate(images):  # ← данных и используем набор данных
        ct = getCt(series_uid)  # ← напрямую. Список серий может быть
                                # ← перетасован, поэтому выполняем
                                # ← сортировку

        for slice_ndx in range(6):
            ct_ndx = slice_ndx * (ct.hu_a.shape[0] - 1) // 5
            sample_tup = dl.dataset.getitem_fullSlice(series_uid, ct_ndx)  # ← Выбор шести
                                                                              # ← равноудаленных
                                                                              # ← срезов по всей КТ

            ct_t, label_t, series_uid, ct_ndx = sample_tup
```

После этого мы подаем `ct_t` в модель. Это очень похоже на то, что происходит в функции `computeBatchLoss`, — подробнее в `p2ch13/training.py`.

Получив `prediction_a`, мы должны создать `image_a`, которое будет содержать отображаемые значения RGB. Мы используем значения `np.float32`, которые должны быть в диапазоне от 0 до 1.

Мы немного схитрим, так как после сложения изображений и масок получим данные в диапазоне от 0 до 2, а затем умножим весь массив на 0,5, чтобы вернуть данные в правильный диапазон (листинг 13.29).

Листинг 13.29. training.py:346, .logImages

```
ct_t[:-1,:,:) /= 2000  # ← Значение интенсивности КТ присваивается
ct_t[:-1,:,:) += 0.5  # ← всем каналам RGB, чтобы получить базовое
                       # ← изображение в оттенках серого

ctSlice_a = ct_t[dl.dataset.contextSlices_count].numpy()

image_a = np.zeros((512, 512, 3), dtype=np.float32)
image_a[:, :, :] = ctSlice_a.reshape((512, 512, 1))  # ←
image_a[:, :, 0] += prediction_a & (1 - label_a)  # ←
image_a[:, :, 0] += (1 - prediction_a) & label_a  # ← Ложноотрицательные — оранжевые
image_a[:, :, 1] += ((1 - prediction_a) & label_a) * 0.5

image_a[:, :, 1] += prediction_a & label_a  # ← Истинно положительные — зеленые
image_a *= 0.5
image_a.clip(0, 1, image_a)
```

Наша цель состоит в том, чтобы получить КТ в градациях серого со сниженной вдвое интенсивностью, наложенную на предполагаемые узелки (или, точнее, узелки-кандидаты) различных цветов. Красный цвет будем использовать для всех неправильных пикселей (ложноположительных и ложноотрицательных). В основном это будут ложные срабатывания, которые нас мало волнуют (поскольку мы сосредоточены на отклике). Выражение `1 - label_a` инвертирует метку, а умножение на `prediction_a` дает нам только предсказанные пиксели, которые не находятся в узелке-кандидате. Для ложноотрицательных результатов к зеленому

добавляется маска со сниженной вдвое интенсивностью, получается оранжевый (1,0 красного и 0,5 зеленого дает как раз это). Каждый правильно предсказанный пиксель внутри узелка становится зеленым без добавления красного.

После этого мы перенормируем наши данные в диапазоне 0...1 и зафиксируем их (на тот случай, если мы начнем выводить дополненные данные и образуются помехи за пределами ожидаемого диапазона КТ). Осталось только сохранить данные в TensorBoard (листинг 13.30).

Листинг 13.30. training.py:361, .logImages

```
writer = getattr(self, mode_str + '_writer')
writer.add_image(
    f'{mode_str}/{series_ndx}_prediction_{slice_ndx}',
    image_a,
    self.totalTrainingSamples_count,
    dataformats='HWC',
)
```

Это очень похоже на вызовы `write.add_scalar`, которые мы видели раньше. Аргумент `data-formats='HWC'` сообщает TensorBoard, что в осях изображения каналы RGB используются в качестве третьей оси. Вспомните, что слои сети часто определяют выходные данные как $B \times C \times H \times W$, и мы могли бы также поместить эти данные непосредственно в TensorBoard, если бы указали `'CHW'`.

Мы также хотим сохранить реальные данные, используемые нами для обучения, чтобы сформировать верхнюю строку наших срезов TensorBoard КТ, которые мы видели ранее на рис. 13.16. Код для этого достаточно похож на то, что мы видели буквально пару минут назад, поэтому пропустим его. Опять же в файле `p2ch13/training.py` можно взглянуть на подробности.

13.6.5. Обновление логирования метрик

Чтобы понимать в процессе обучения, как идут дела, мы вычисляем метрики для каждой эпохи, а именно количество истинно положительных, ложноотрицательных и ложноположительных результатов. Это и делает приведенный ниже код (листинг 13.31). Здесь не будет ничего особо удивительного.

Листинг 13.31. training.py:400, .logMetrics

```
sum_a = metrics_a.sum(axis=1)
allLabel_count = sum_a[METRICS_TP_NDX] + sum_a[METRICS_FN_NDX]
metrics_dict['percent_all/tp'] = \
    sum_a[METRICS_TP_NDX] / (allLabel_count or 1) * 100
metrics_dict['percent_all/fn'] = \
    sum_a[METRICS_FN_NDX] / (allLabel_count or 1) * 100
metrics_dict['percent_all/fp'] = \
    sum_a[METRICS_FP_NDX] / (allLabel_count or 1) * 100
```

Может получиться больше 100 %, поскольку мы сравниваем с общим количеством пикселей, помеченных как узелки-кандидаты, а это лишь крошечная часть каждого изображения

Нужно будет оценивать полученные модели, чтобы определить, является ли конкретный тренировочный прогон лучшим из всех, которые уже были сделаны. В главе 12 мы отметили, что для оценки качества модели используется метрика F1, но здесь у нас другие цели. Мы должны получить как можно более высокий отклик, поскольку нельзя классифицировать потенциальный узелок, если не найти его!

Параметр отклика поможет нам определить лучшую модель. Метрика F1 в целом тоже подходит¹, но мы просто хотим получить как можно более высокий отклик. Отсеивание ложноположительных результатов ляжет на плечи модели классификации (листинг 13.32).

Листинг 13.32. training.py:393, .logMetrics

```
def logMetrics(self, epoch_ndx, mode_str, metrics_t):
    # ... строка 453
    score = metrics_dict['pr/recall']

    return score
```

Когда мы добавим аналогичный код в цикл обучения классификации в следующей главе, мы будем использовать метрику F1.

Вернувшись к основному циклу обучения, мы будем отслеживать параметр `best_score`, который ранее получился в прогоне. Сохраняя модель, мы включаем флаг, указывающий, является ли данный результат лучшим, который мы видели до сих пор. Напомним материал подраздела 13.6.4: мы вызываем функцию `doValidation` только для первой, а затем для каждой пятой эпохи. То есть будем проверять только лучшие результаты. Это не должно быть проблемой, но об этом нужно помнить, если вам нужно отладить что-то происходящее в эпоху 7. Мы делаем данную проверку непосредственно перед сохранением изображений (листинг 13.33).

Листинг 13.33. training.py:210, SegmentationTrainingApp.main

```
def main(self):
    best_score = 0.0
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        # если требуется проверка
        # ... строка 233
        valMetrics_t = self.doValidation(epoch_ndx, val_d1)
        score = self.logMetrics(epoch_ndx, 'val', valMetrics_t)
        best_score = max(score, best_score)

        self.saveModel('seg', epoch_ndx, score == best_score)
```

Цикл по эпохам, который мы уже видели

Вычисление метрики. Как уже говорилось, мы берем отклик

Теперь осталось только записать `saveModel`. Третий параметр указывает, является ли эта модель лучшей

Посмотрим, как модель сохраняется на диск.

¹ Хотя и это довольно шаткое определение.

13.6.6. Сохранение модели

PyTorch упрощает процедуру сохранения модели на диск. «Под капотом» функции `torch.save` используется стандартная библиотека Python `pickle`, которой можно напрямую передать экземпляр модели, и он сохранится правильно. Однако это не считается идеальным способом сохранения нашей модели, поскольку он лишает нас некой гибкости.

Вместо этого мы сохраним только *параметры* модели. Так мы получим возможность загружать эти параметры в любую модель, способную принимать параметры той же формы, даже если класс не соответствует модели, в которой были сохранены эти параметры. Подход с сохранением только параметров позволяет нам повторно использовать и смешивать модели более гибко, чем если бы мы сохраняли сразу всю модель.

Мы можем получить параметры нашей модели, используя функцию `model.state_dict()` (листинг 13.34).

Листинг 13.34. training.py:480, `saveModel`

```
def saveModel(self, type_str, epoch_ndx, isBest=False):
    # ... строка 496
    model = self.segmentation_model
    if isinstance(model, torch.nn.DataParallel):
        model = model.module  # ← Избавляется от оболочки DataParallel, если она существует
    state = {
        'sys_argv': sys.argv,
        'time': str(datetime.datetime.now()),
        'model_state': model.state_dict(),  # ← Важная часть
        'model_name': type(model).__name__,
        'optimizer_state': self.optimizer.state_dict(),  # ← Сохранение момента
        'optimizer_name': type(self.optimizer).__name__,
        'epoch': epoch_ndx,
        'totalTrainingSamples_count': self.totalTrainingSamples_count,
    }
    torch.save(state, file_path)
```

В переменную `file_path` нужно положить нечто вроде `data-unversioned/part2/models/p2ch13/seg_2019-07-10_02.17.22_ch12.50000.state`. Часть `.50000` соответствует количеству обучающих элементов данных, которые мы уже передали модели, а остальные части пути очевидны.

Если текущая модель дает лучший результат из всех, что мы видели до сих пор, мы сохраняем вторую копию `state` с именем файла `.best.state`. Позже она может быть перезаписана другой версией модели с более высокой метрикой. Сосредоточив внимание на файле с лучшим результатом, мы можем отвлечь клиентов нашей обученной модели от подробностей происходящего в каждую

эпоху обучения (при условии, что используется достаточно качественная метрика) (листинг 13.35).

Листинг 13.35. training.py:514, .saveModel

```
if isBest:
    best_path = os.path.join(
        'data-unversioned', 'part2', 'models',
        self.cli_args.tb_prefix,
        f'{type_str}_{self.time_str}_{self.cli_args.comment}.best.state')
    shutil.copyfile(file_path, best_path)

    log.info("Saved model params to {}".format(best_path))

with open(file_path, 'rb') as f:
    log.info("SHA1: " + hashlib.shal(f.read()).hexdigest())
```

СОВЕТ

Сохранив еще и состояние оптимизатора, можно было бы реализовать возможность возобновить обучение. Здесь мы этого не приводим, но в целом методика может быть полезной, если вдруг вы не можете довести процесс до конца. Подробности о загрузке модели и оптимизатора для перезапуска обучения можно найти в официальной документации (https://pytorch.org/tutorials/beginner/saving_loading_models.html).

Мы также выводим значение SHA1 только что сохраненной модели. Подобно `sys.argv` и отметке времени, которую мы помещаем в словарь состояния, это может помочь нам отладить именно ту модель, с которой мы работаем, если позже возникнет путаница (например, файл будет переименован неправильно).

В следующей главе мы обновим сценарий обучения классификатора, чтобы он умел так же сохранять модель классификации. Для диагностики КТ нам понадобятся обе модели.

13.7. РЕЗУЛЬТАТЫ

Теперь, когда мы внесли все нужные изменения в код, мы подошли к этапу 3 на рис. 13.17. Пришло время запустить команду `python -m p2ch13.training --epochs 20 --augmented final_seg`. Посмотрим, какие получатся результаты!

Ниже приведены метрики обучения в случае, если мы ограничимся эпохами, для которых у нас есть проверочные метрики (мы рассмотрим их далее):

В этих строках нас особенно интересует метрика F1, которая растет. Это хорошо!		Истинно положительные результаты движутся вверх. Отлично! Ложноположительные и ложноотрицательные снижаются	
	E1 trn 0.5235 loss, 0.2276 precision, 0.9381 recall, 0.3663 f1 score		←
	E1 trn_all 0.5235 loss, 93.8% tp, 6.2% fn, 318.4% fp		←
	...		
→	E5 trn 0.2537 loss, 0.5652 precision, 0.9377 recall, 0.7053 f1 score		←
	E5 trn_all 0.2537 loss, 93.8% tp, 6.2% fn, 72.1% fp		←
	...		
→	E10 trn 0.2335 loss, 0.6011 precision, 0.9459 recall, 0.7351 f1 score		←
	E10 trn_all 0.2335 loss, 94.6% tp, 5.4% fn, 62.8% fp		←
	...		
→	E15 trn 0.2226 loss, 0.6234 precision, 0.9536 recall, 0.7540 f1 score		←
	E15 trn_all 0.2226 loss, 95.4% tp, <2> 4.6% fn, 57.6% fp		←
	...		
→	E20 trn 0.2149 loss, 0.6368 precision, 0.9584 recall, 0.7652 f1 score		←
	E20 trn_all 0.2149 loss, 95.8% tp, <2> 4.2% fn, 54.7% fp		←

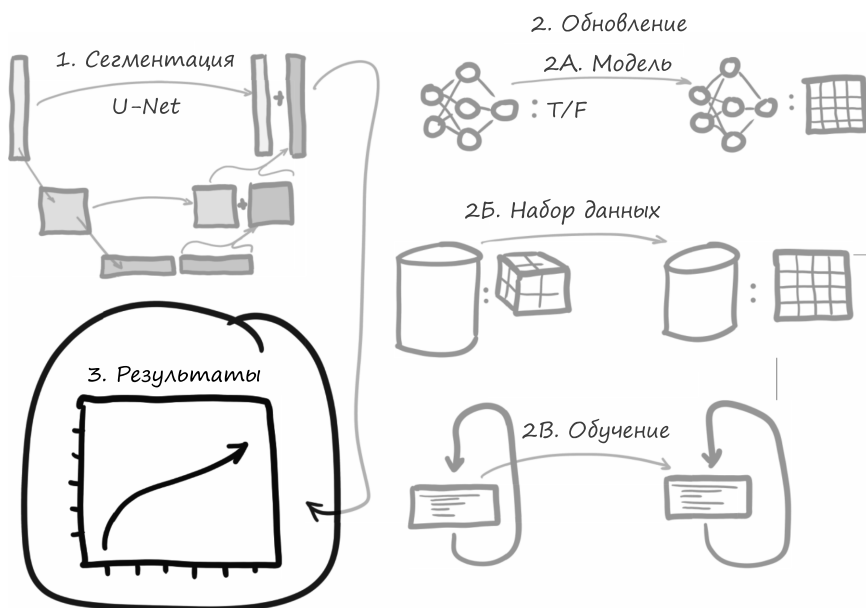


Рис. 13.17. План этой главы — результаты после обучения

В целом результат выглядит достаточно хорошо. Истинно положительные результаты и метрика F1 имеют тенденцию к росту, ложноположительные и отрицательные — к снижению. Именно это нам и нужно было! Проверочные метрики скажут нам, насколько качественные эти результаты. Имейте в виду: поскольку мы тренируемся на фрагментах 64×64 , но проверку выполняем на целых срезах КТ 512×512 , у нас почти наверняка будут совершенно другие соотношения ИП:ЛО:ЛП. Посмотрим:

Самый высокий показатель истинно положительных (отличный).

Обратите внимание, что процент у них такой же, как и отклик.

А вот ложноположительных получилось 4495 % — многовато

```

E1 val      0.9441 loss, 0.0219 precision, 0.8131 recall, 0.0426 f1 score
E1 val_all  0.9441 loss, 81.3% tp, 18.7% fn, 3637.5% fp

E5 val      0.9009 loss, 0.0332 precision, 0.8397 recall, 0.0639 f1 score
E5 val_all  0.9009 loss, 84.0% tp, 16.0% fn, 2443.0% fp

E10 val     0.9518 loss, 0.0184 precision, 0.8423 recall, 0.0360 f1 score
E10 val_all 0.9518 loss, 84.2% tp, 15.8% fn, 4495.0% fp

E15 val     0.8100 loss, 0.0610 precision, 0.7792 recall, 0.1132 f1 score
E15 val_all 0.8100 loss, 77.9% tp, 22.1% fn, 1198.7% fp

E20 val     0.8602 loss, 0.0427 precision, 0.7691 recall, 0.0809 f1 score
E20 val_all 0.8602 loss, 76.9% tp, 23.1% fn, 1723.9% fp

```

Откуда взялось более 4000 % ложноположительных результатов? Вообще, это ожидаемо. Область проверочного среза составляет 2^{18} пикселей ($512 = 2^9$), а обучающий фрагмент — всего 2^{12} пикселей. Это значит, что мы проверяем поверхность среза, которая в $2^6 = 64$ раза больше обучающего фрагмента! Так что, когда ложноположительная метрика получается в 64 раза больше, это не удивляет. Притом доля истинно положительных результатов существенно не изменится, поскольку все они были бы включены в выборку 64×64 , на которой выполнялось обучение. Кроме того, данная ситуация приводит к очень низкой точности и, следовательно, к низкой метрике F1. Это естественный результат того, как мы структурировали обучение и проверку, так что волноваться тут не о чем.

Проблема заключается в отклике (и, следовательно, в доле истинно положительных результатов). Показатель отклика выходит на плато между эпохами 5 и 10, а затем начинает падать. Совершенно очевидно, что модель быстро начинает переобучаться; это дополнительно видно на рис. 13.18, где отклик обучения продолжает расти, а отклик проверки уменьшается после 3 миллионов элементов данных. Именно так мы определяли переобучение в главе 5, в частности на рис. 5.14.

ПРИМЕЧАНИЕ

Всегда помните, что TensorBoard по умолчанию сглаживает строки данных. Более светлая призрачная линия за сплошным цветом соответствует необработанным значениям.

Архитектура U-Net обладает высокой пропускной способностью, и даже с нашим уменьшенным количеством фильтров и уровней глубины она способна довольно быстро запомнить тренировочный набор. Одним из преимуществ архитектуры является то, что модель обучается довольно быстро!

В задаче сегментации наш главный параметр — отклик, а проблемы с точностью мы перекладываем на плечи моделей классификации. Снижение количества

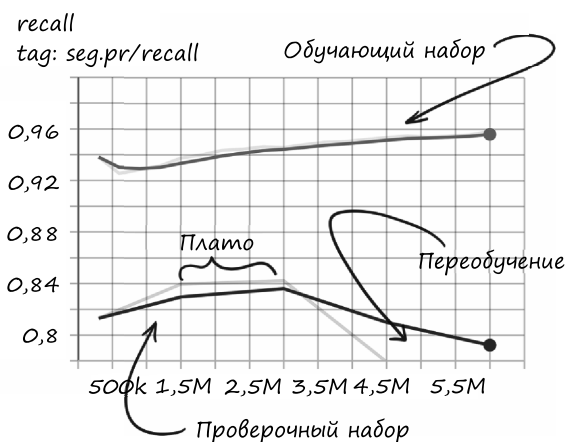


Рис. 13.18. Отклик проверочного набора с признаками переобучения — отклик снижается после эпохи 10 (3 миллиона точек данных)

ложноположительных результатов — именно то, для чего и нужны модели классификации! Эта несколько искаженная ситуация означает, что оценить нашу модель окажется сложнее, чем нам хотелось бы. Мы могли бы использовать метрику F2, которая больше внимания уделяет отклику (или F5, или F10...), но тогда пришлось бы выбрать достаточно высокое значение N , чтобы почти полностью обесценить точность. Мы пропустим промежуточные этапы и просто оценим нашу модель по отклику, а затем посмотрим своим взглядом эксперта, не является ли данный обучающий прогон патологическим. Поскольку во время обучения мы используем потерю Дайса, а не само значение отклика, все должно работать.

Это один из тех случаев, когда в ход идет хитрость, поскольку мы (авторы) уже провели обучение и оценку для главы 14 и знаем, чем кончится дело. Не существует хорошего способа оценить всю ситуацию заранее и *быть уверенными*, что все работает хорошо. Обоснованные догадки полезны, но они не заменят реальных экспериментов.

В нынешнем виде наши результаты достаточно хороши, чтобы их можно было использовать в будущем, даже если у метрик будут экстремальные значения. Мы на шаг ближе к завершению нашего проекта!

13.8. ИТОГИ ГЛАВЫ

В этой главе мы рассмотрели новый способ структурирования моделей для попиксельной сегментации, ввели готовую и уже проверенную в подобных задачах архитектуру U-Net, а также адаптировали реализацию для собственного использования. Кроме того, мы изменили исходный набор данных, чтобы они лучше подходили новой модели. Для этого мы выделили в изображениях

небольшие фрагменты для обучения и проверочные срезы. Теперь мы можем сохранять изображения в `TensorBoard` в обучающем цикле, а дополнение данных мы перенесли в отдельную модель, которая может работать на ГП. Наконец, мы проанализировали результаты обучения и пришли к выводу, что, даже несмотря на не вполне ожидаемое высокое количество ложноположительных результатов, общая картина оказалась приемлемой, учитывая наши требования к ним на уровне проекта в целом. В главе 14 мы объединим модели, которые мы написали, в связанное сквозное целое.

13.9. УПРАЖНЕНИЯ

1. Реализуйте обертку для модели, выполняющую дополнение (подобную тому, что мы использовали для обучения сегментации) для модели классификации.
 - А. На какие компромиссы вам пришлось пойти?
 - Б. Как это повлияло на скорость обучения?
2. Измените реализацию `Dataset` сегментации, чтобы получить трехстороннее разделение обучающих, проверочных и тестовых наборов.
 - А. Какую часть данных вы использовали для тестового набора?
 - Б. Соответствуют ли результаты тестового и проверочного наборов?
 - В. Насколько сильно страдает обучение при использовании меньшего тренировочного набора?
3. Попробуйте заставить сегментировать злокачественные и доброкачественные новообразования, а не просто определять наличие узелка.
 - А. Что должно измениться в отчетах? А в генерации изображений?
 - Б. Какие результаты у вас получились? Достаточно ли хороша сегментация, чтобы пропустить этап классификации?
4. Можно ли обучить модель одновременно на фрагментах размером 64×64 и цельных срезах КТ?¹
5. Удастся ли вам найти дополнительные источники данных, кроме LUNA (или LIDC)?

¹ Подсказка: каждый кортеж элементов данных, которые подаются вместе, должен иметь одинаковую форму для каждого соответствующего тензора, но в следующем пакете могут быть уже другие кортежи с другими формами.

13.10. РЕЗЮМЕ

- Модель сегментации определяет принадлежность отдельных пикселей или вокселей к определенному классу. Этим сегментация отличается от классификации, которая работает на уровне всего изображения.
- Архитектура U-Net в свое время была революционной архитектурой для задач сегментации.
- Используя сегментацию с последующей классификацией, мы можем реализовать обнаружение объектов, не предъявляя слишком серьезных требований к данным и вычислениям.
- Наивные подходы к 3D-сегментации могут привести к быстрому использованию слишком большого объема оперативной памяти у ГП текущего поколения. Ограничение объема данных, передаваемых модели, может помочь снизить затраты оперативной памяти.
- Можно обучать модель сегментации на кадрированных изображениях, одновременно проверяя полноценные срезы изображения. Эта гибкость может оказаться полезна с точки зрения балансировки классов.
- Взвешивание потерь позволяет выделить значимость потерь определенных классов или подмножеств обучающих данных, чтобы отклонить поведение модели в нужную сторону. Этот механизм, вкуче с балансировкой классов, является полезным инструментом для регулировки производительности модели.
- TensorBoard может отображать 2D-изображения, созданные во время обучения, и сохранять историю изменений моделей в ходе обучения. Эту возможность используют для визуального отслеживания изменений в выходных данных модели по мере прохождения обучения.
- Параметры модели можно сохранить на диск и загрузить обратно, чтобы воссоздать ранее сохраненную модель. Точная реализация модели при этом может даже быть другой, если параметры старой и новой модели в точности соответствуют друг другу.

14

Сквозной анализ узелков и дальнейшее развитие проекта

В этой главе

- ✓ Объединение моделей сегментации и классификации.
- ✓ Тонкая настройка сети для новой задачи.
- ✓ Добавление в TensorBoard гистограммы и других метрик.
- ✓ Переход от переобучения к обобщению.

На протяжении нескольких последних глав мы создали очень много систем, из которых состоит конечный проект. Мы реализовали загрузку данных, создали и усовершенствовали классификаторы для узелков-кандидатов, обучили модели сегментации для поиска этих кандидатов, создали вспомогательную инфраструктуру, необходимую для обучения и оценки эффективности, а также научились сохранять результаты обучения на диск. Теперь пришло время объединить все имеющиеся компоненты в единое целое и создать из них полную цепь проекта. Начнем автоматически обнаруживать рак!

14.1. ФИНИШНАЯ ПРЯМАЯ

Чтобы иметь представление об оставшейся работе, можно взглянуть на рис. 14.1. На этапе 3 (группировка) сказано, что нам все еще нужно построить мост между моделью сегментации из главы 13 и классификатором из главы 12, который

определяет, действительно ли образование, обнаруженное сетью сегментации, является узелком. Справа показан пятый (анализ узла и диагностика) и последний этап на пути к нашей финальной цели: определить, является ли узелок раком. Это тоже задача классификации, но в образовательных целях для ее решения мы воспользуемся другим подходом, взяв за основу уже имеющийся у нас классификатор узелков.

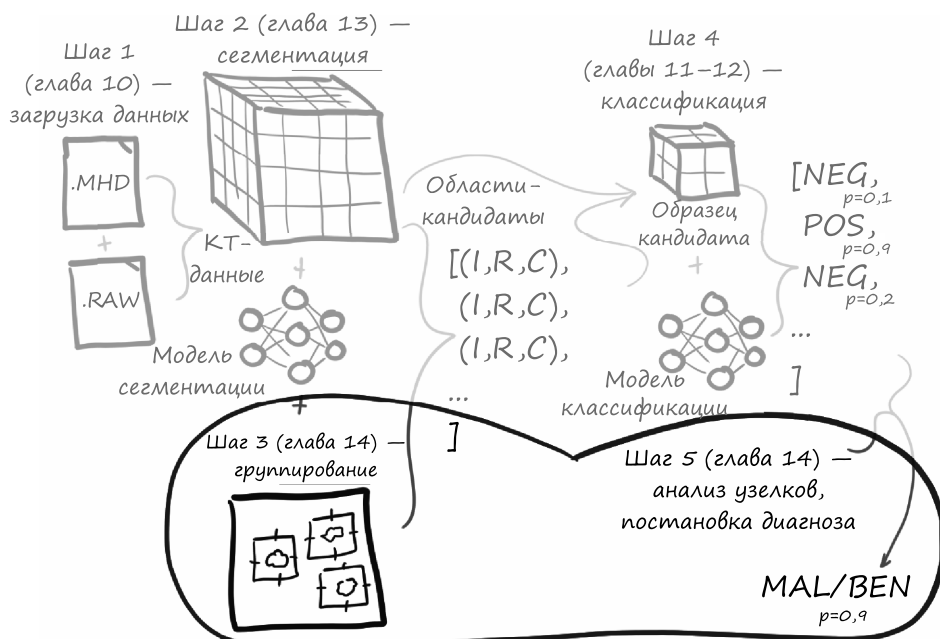


Рис. 14.1. Комплексный проект по обнаружению рака легких. Выделены основные темы данной главы: этап 3 — группировка, этап 5 — анализ узлов

Разумеется, на рис. 14.1 показано весьма упрощенное изображение, которое выносит за скобки очень много деталей. Немного увеличим рис. 14.2 и посмотрим, что нам осталось сделать.

Как видите, остались три важные задачи. Каждый пункт в приведенном ниже списке соответствует пункту на рис. 14.2.

1. *Создание узелков-кандидатов.* Это этап 3 в общем проекте. Здесь выполняются три задачи.
 - А. *Сегментация.* Модель сегментации из главы 13 предскажет, интересен ли нам данный конкретный пиксель, а именно является ли он частью узелка. Сегментация выполняется для каждого 2D-среза, а затем 2D-результаты

объединяются в трехмерный массив вокселей, содержащий прогнозы кандидатов на узелки.

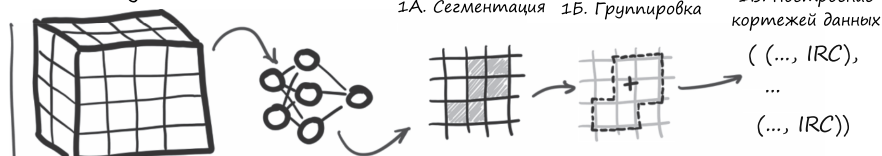
Б. *Группировка*. Мы группируем отдельные воксели в узелки-кандидаты с помощью порогового значения и объединения стоящих рядом помеченных вокселей.

В. *Построение кортежей данных*. Каждый узелок-кандидат будет использоваться для создания элемента данных для классификации. В частности, нам нужно получить координаты (индекс, строка, столбец) центра узелка.

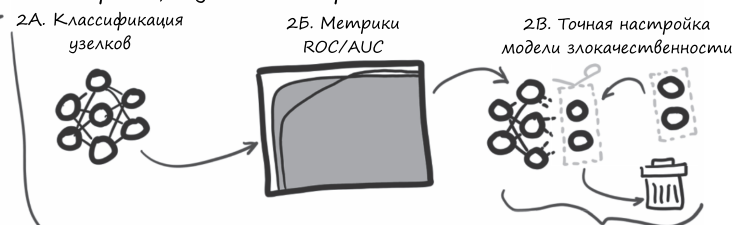
Как только мы это реализуем, у нас получится приложение, которое берет необработанную компьютерную томографию пациента и создает список обнаруженных на ней кандидатов в узелки. Составление такого списка — задача в конкурсе LUNA. Если бы наш проект использовался в клинических условиях (а мы еще раз подчеркиваем, что делать это в нашем случае не надо!), данный список узелков был бы передан врачу для более тщательного анализа.

2. *Классификация узелков и определение злокачественности*. Мы возьмем только что созданные узелки-кандидаты и передадим их на этап классификации кандидатов, который реализовали в главе 12, а затем определим злокачественные новообразования среди кандидатов, помеченных как узелки.

1. Создание узелков-кандидатов



2. Классификация узелков и определение злокачественности



3. Сквозное обнаружение

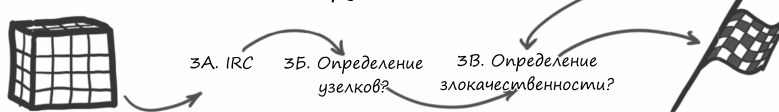


Рис. 14.2. Более подробная схема оставшейся работы

- А. *Классификация узелков.* Каждый узелок-кандидат, полученный после сегментации и группировки, пройдет классификацию, чтобы определить, действительно ли это узелок. Так появится возможность отфильтровать множество нормальных анатомических структур, помеченных моделью сегментации.
- Б. *Метрики ROC/AUC.* Прежде чем перейти к последнему этапу классификации, мы определим новые метрики для изучения эффективности моделей классификации, а также установим базовый показатель, с которым можно сравнивать классификаторы злокачественных новообразований.
- В. *Точная настройка модели злокачественности.* Имея под рукой метрики, мы определим модель специально для классификации доброкачественных и злокачественных узелков, обучим ее и оценим ее работу. Мы будем проводить обучение путем тонкой настройки, то есть отбирать из модели некие весовые коэффициенты и заменять их новыми значениями, которые затем адаптируем к нашей новой задаче.

В этот момент наша конечная цель будет совсем рядом, и мы сможем классифицировать узлы на доброкачественные и злокачественные, а затем ставить диагноз, имея на входе лишь КТ. Опять же диагностика рака легких в реальном мире — нечто большее, чем просмотр компьютерной томографии, поэтому в нашем случае постановка диагноза — скорее проба того, как далеко мы можем продвинуться, используя только данные глубокого обучения и визуализации.

- 3. *Сквозное обнаружение.* Наконец, мы соберем все воедино и выйдем на финишную прямую, объединив компоненты в комплексное решение, которое может посмотреть на КТ и ответить на вопрос «Есть ли в легких злокачественные узелки?».
- А. *IRC.* Мы будем сегментировать КТ, чтобы получить узелки-кандидаты для классификации.
- Б. *Определение узелков.* Мы проведем классификацию узелков, чтобы определить, нужно ли отправлять тот или иной узелок в классификатор злокачественности.
- В. *Определение злокачественности.* Выполним классификацию злокачественных новообразований среди узелков, прошедших первый классификатор, чтобы определить, есть ли у пациента рак.

Короче, работы у нас предостаточно. К финишу!

ПРИМЕЧАНИЕ

Как и в предыдущей главе, в тексте мы подробно обсудим только ключевые концепции и опустим код повторяющихся, утомительных или очевидных частей. Полную информацию можно найти в репозитории кода книги.

14.2. НЕЗАВИСИМОСТЬ ПРОВЕРОЧНОГО НАБОРА

У нас есть риск совершить тонкую, но фатальную ошибку, которую нужно обсудить и обойти. Существует потенциальный риск утечки данных из обучающего набора в проверочный! В каждой из моделей сегментации и классификации мы выполняли разделение данных на обучающий набор и независимый проверочный набор, отобрав каждый десятый пример для проверки, а остальные — для обучения.

Однако разделение для модели классификации выполнялось по списку узелков, а разделение для модели сегментации — по списку КТ. В результате, вероятно, некоторые узелки из проверочного набора сегментации попали в обучающий набор модели классификации и наоборот. Этого необходимо избежать! Если не исправить такую ситуацию, то показатели эффективности модели могут оказаться искусственно завышенными по сравнению с тем, что мы получим на независимом наборе данных. Это явление называется *утечкой*, и оно делает всю процедуру проверки недействительной.

Устранить эту возможную утечку данных можно следующим образом: переработать набор данных классификации, чтобы он тоже работал на уровне компьютерной томографии, как и в задаче сегментации в главе 13. Затем нам нужно переобучить модель классификации на новом наборе данных. К тому же мы все равно не сохранили классификационную модель ранее, поэтому повторное обучение все равно пришлось бы выполнить.

Из этого следует извлечь важный урок: при определении проверочного набора нужно смотреть на весь процесс целиком. Вероятно, самый простой способ сделать это (для важных наборов данных так и происходит) — реализовать разделение данных проверки как можно более явно, например используя два отдельных каталога для обучающих и проверочных данных, а затем придерживаться этого разделения на протяжении всего проекта. Когда вам нужно выполнить разделение заново (скажем, если требуется добавить стратификацию набора данных, разделенного по какому-либо критерию), нужно переобучить ваши модели с новым набором данных.

Итак, мы взяли `LunaDataset` из глав 10–12 и скопировали получение списка кандидатов и разделение его на тестовые и проверочные наборы данных из `Luna2dSegmentationDataset` в главе 13. Это чисто механическая процедура, не несущая новой информации (вы ведь уже профессионал в работе с данными), так что мы не будем подробно показывать код.

Переобучим модель классификации, повторно запустив обучение классификатора¹:

```
$ python3 -m p2ch14.training --num-workers=4 --epochs 100 nodule-nonodule
```

¹ Вы также можете использовать документ `Jupiter p2_run_everything`.

Спустя 100 эпох мы достигаем точности около 95 % для положительных образцов и 99 % для отрицательных. Поскольку потери при проверке не имеют тенденции к увеличению, можно продлить обучение, чтобы увидеть, продолжает ли улучшаться ситуация.

После 90 эпох мы достигаем максимального значения F1 и имеем точность проверки 99,2 %, хотя из них лишь 92,8 % касаются реальных узелков. Мы возьмем эту модель, даже несмотря на то, что можно было бы несколько снизить общую точность ради более высокой точности определения злокачественных узелков (был момент, когда модель достигла точности 95,4 % для реальных узелков при общей точности 98,9 %). Нам этого будет достаточно, так что можно соединять модели.

14.3. ОБЪЕДИНЕНИЕ СЕГМЕНТАЦИИ КТ И КЛАССИФИКАЦИИ УЗЕЛКОВ-КАНДИДАТОВ

Теперь, когда у нас есть модель сегментации из главы 13 и модель классификации, которую мы обучили только что в предыдущем разделе, на рис. 14.3 этапы 1А, 1Б и 1В демонстрируют, что пришло время написать код, который преобразует выходные данные сегментации в кортежи данных. Нам нужно выполнить *группировку*: то есть определить пунктирный контур, как на этапе 1Б на рис. 14.3. Входные данные — это *сегментация*, полученная моделью сегментации на этапе 1А. Нам нужно реализовать этап 1В, то есть найти координаты центра масс каждой группы помеченных вокселей: индекс, строка и столбец точки, отмеченной крестиком на этапе 1Б. Эти данные в виде кортежей будут являться выходными.

Запуск моделей, естественно, будет очень похож на то, что мы делали ранее во время обучения и проверки (особенно проверки). Разница здесь заключается в цикле, который перебирает КТ. У каждой КТ мы сегментируем *каждый* срез, а затем берем весь сегментированный вывод в качестве входных данных для группировки. Результат группировки будет передан в классификатор узелков, а узлы, выжившие после такой классификации, будут переданы в классификатор злокачественных новообразований. Все это реализуется с помощью следующего внешнего цикла по КТ, который для каждой КТ выполняет сегментацию, группировку и классификацию кандидатов и предоставляет классификации для дальнейшей обработки (листинг 14.1).

Листинг 14.1. nodule_analysis.py:324, NoduleAnalysisApp.main

```
for _, series_uid in series_iter:  ← Перебор по UID
    ct = getCt(series_uid)  ← Получение КТ (этап 1 на большой схеме)
    mask_a = self.segmentCt(ct, series_uid)  ← Запуск модели сегментации (этап 2)

    candidateInfo_list = self.groupSegmentationOutput(  ← Группировка отмеченных
        series_uid, ct, mask_a)                        вокселей на выходе (этап 3)
    classifications_list = self.classifyCandidates(  ← Запуск классификатора узелков
        ct, candidateInfo_list)                      (этап 4)
```

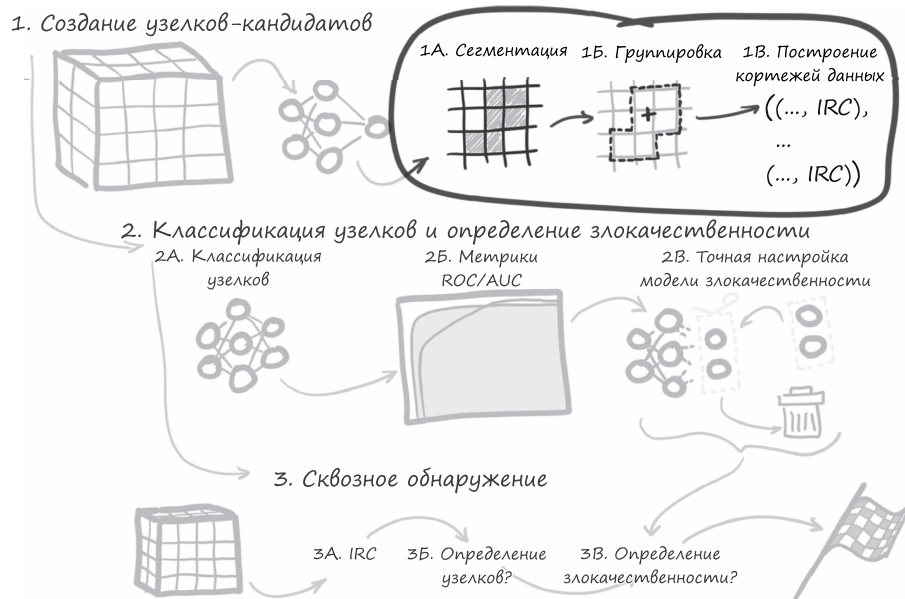


Рис. 14.3. План этой главы. Выделен этап группировки сегментированных вокселей в кандидаты на узелки

Методы `segmentCt`, `groupSegmentationOutput` и `classifyCandidates` мы подробно рассмотрим ниже.

14.3.1. Сегментация

Прежде всего нам нужно выполнить сегментацию каждого среза всей компьютерной томографии. Поскольку КТ данного пациента передается срез за срезом, мы строим `Dataset`, который загружает КТ с одним `series_uid` и возвращает каждый ее срез, по одному на вызов `__getitem__`.

ПРИМЕЧАНИЕ

Этап сегментации может занять довольно много времени при выполнении на процессоре. Мы не делаем на этом акцент, но код будет использовать ГП, если тот будет доступен.

Помимо большего объема входных данных, основное различие заключается в том, что происходит с выходными данными дальше. Напомним: на выходе получается массив попиксельных вероятностей (то есть значений в диапазоне от 0 до 1) того, что данный пиксель является частью узелка. Перебирая срезы, мы собираем прогнозы по срезам в массиве масок, который имеет ту же форму,

что и входные данные КТ. После этого мы выставляем пороговое значение, чтобы получить двоичный массив. Мы будем использовать пороговое значение 0.5, но можно было бы и поэкспериментировать с ним, чтобы получить больше истинно положительных результатов в обмен на увеличение ложноотрицательных.

Кроме того, мы включим небольшую процедуру очистки с помощью операции эрозии из библиотеки `scipy.ndimage.morphology`. Эта операция удаляет один слой граничных вокселей и сохраняет только внутренние — те, у которых также помечены все восемь соседних вокселей. Это позволяет уменьшить отмеченную область и удалить из данных маленьких кандидатов (меньше чем $3 \times 3 \times 3$ вокселя) на исчезновение. Добавим к этому цикл, перебирающий загрузчик данных, которому мы даем указание передать все срезы с одной КТ. Получается следующее (листинг 14.2).

Листинг 14.2. `nodule_analysis.py:384, .segmentCt`

```

def segmentCt(self, ct, series_uid):
    with torch.no_grad():
        output_a = np.zeros_like(ct.hu_a, dtype=np.float32)
        seg_dl = self.initSegmentationDl(series_uid) #
        for input_t, _, _, slice_ndx_list in seg_dl:

            input_g = input_t.to(self.device)
            prediction_g = self.seg_model(input_g)

            for i, slice_ndx in enumerate(slice_ndx_list):
                output_a[slice_ndx] = prediction_g[i].cpu().numpy()

            mask_a = output_a > 0.5
            mask_a = morphology.binary_erosion(mask_a, iterations=1)

    return mask_a

```

Градиенты здесь не нужны, поэтому график не строим

Этот массив будет содержать данные: числа с плавающей запятой, обозначающие вероятности

Мы получаем загрузчик данных, который позволяет перебирать КТ пакетами

После перемещения входных данных в ГП...

...и копируем каждый элемент в выходной массив

Применяем к вероятностям пороговое значение, чтобы получить двоичный вывод, а затем применяем двоичную эрозию для очистки

...мы запускаем модель сегментации...

Это было достаточно просто, но теперь нам нужно придумать группировку.

14.3.2. Группировка вокселей в узелки-кандидаты

Для группировки вокселей в узелки-кандидаты по фрагментам для дальнейшей классификации мы используем простой алгоритм связанных компонентов. Этот алгоритм помечает связанные компоненты, и мы реализуем его с помощью `scipy.ndimage.measurements.label`. Функция `label` берет все ненулевые пиксели, стоящие рядом с другим ненулевым пикселем, и помечает их как принадлежащие к той же группе. Поскольку выходные данные модели сегментации в основном

состоят из очень близко расположенных друг к другу пикселей, этот подход нас устраивает (листинг 14.3).

Листинг 14.3. nodule_analysis.py:401

```
def groupSegmentationOutput(self, series_uid, ct, clean_a):
    candidateLabel_a, candidate_count = measurements.label(clean_a)
    centerIrc_list = measurements.center_of_mass(
        ct.hu_a.clip(-1000, 1000) + 1001,
        labels=candidateLabel_a,
        index=np.arange(1, candidate_count+1),
    )
```

Присваиваем каждому вокселю метку группы, к которой он принадлежит

Получаем центр масс для каждой группы: индекс, строка, координаты столбца

Выходной массив `candidateLabel_a` получается такой же формы, как `clean_a`, который мы использовали для ввода, но содержит значения 0 на месте фоновых вокселей и целочисленные метки 1, 2... у каждого из связанных блоков вокселей, из которых состоит узелок-кандидат. Обратите внимание: метки здесь — не то же самое, что метки в классификации! Здесь они просто говорят: «Эта кучка вокселей — кучка 1, а вот эта — кучка 2 и т. д.».

В SciPy также есть функция для получения центров масс кандидатов в узелки: `scipy.ndimage.measurements.center_of_mass`. Она принимает на вход массив с плотностью вокселей, целочисленные метки из вызванной нами функции `label` и список тех меток, для которых нужно вычислить центр. Чтобы соответствовать ожиданиям функции, что масса неотрицательна, мы смещаем (обрезаем) `ct.hu_a` на 1.001. Обратите внимание: в результате у всех помеченных вокселей получается некий вес, поскольку мы зафиксировали самое низкое значение для воздуха равным -1000 HU в исходных единицах КТ (листинг 14.4).

Листинг 14.4. nodule_analysis.py:409

```
candidateInfo_list = []
for i, center_irc in enumerate(centerIrc_list):
    center_xyz = irc2xyz(
        center_irc,
        ct.origin_xyz,
        ct.vxSize_xyz,
        ct.direction_a,
    )

    candidateInfo_tup = \
        CandidateInfoTuple(False, False, False, 0.0, series_uid, center_xyz)
    candidateInfo_list.append(candidateInfo_tup)

return candidateInfo_list
```

Преобразуем координаты вокселя в реальные координаты пациента

Составляем кортеж с информацией о кандидате и добавляем его в список обнаруженных

На выходе получается список из трех массивов (по одному для индекса, строки и столбца) той же длины, что и `candidate_count`. Мы можем использовать эти данные для заполнения списка экземпляров `candidateInfo_tup`. Мы уже привязались к этой небольшой структуре данных, поэтому помещаем результаты в тот же список, который использовали, начиная с главы 10. Поскольку у нас нет подходящих данных для первых четырех значений (`isNodule_bool`, `hasAnnotation_bool`, `isMal_bool`, и `diameter_mm`), мы вставляем значения-заполнители подходящего типа. Затем конвертируем координаты из вокселей в физические координаты в цикле, создавая список. Может показаться немного глупым отделять координаты от массива с индексами, строками и столбцами, но весь код, который потребляет экземпляры `candidateInfo_tup`, ожидает `center_xyz`, а не `center_irc`. Мы получили бы совершенно неправильные результаты, если бы попытались заменить одно другим!

Отлично, мы осилили этап 3 и получили расположение узелков по вокселям! Теперь можно вырезать из данных кандидатов в узелки и передать их классификатору, чтобы отсеять еще несколько ложноположительных результатов.

14.3.3. Узелок или не узелок? Классификация и снижение числа ложноположительных результатов

В начале части II мы говорили о работе рентгенолога, просматривающего компьютерную томографию на предмет признаков рака:

«В текущем варианте работа по анализу данных должна выполняться высококвалифицированными специалистами, требует кропотливого внимания к деталям, и в большинстве случаев рак не обнаруживается.

Для человека такая работа сродни тому, как если бы вас поставили перед сотней стогов сена и сказали: “Определите, в каких из них есть иголка”».

Мы потратили немало времени и сил на поиск иголок. На секунду отвлечемся и обсудим сено, взглянув на рис. 14.4. Наша задача — максимально помочь рентгенологу, чтобы он мог перенаправить свое экспертное внимание туда, где оно принесет наибольшую пользу.

Посмотрим, сколько данных отбрасывается на каждом этапе сквозного проекта. Стрелками на рис. 14.4 обозначен поток данных через весь проект от необработанных вокселей КТ до окончательного определения злокачественности. Каждая стрелка, оканчивающаяся крестиком, указывает на данные, отброшенные на предыдущем этапе. Стрелка, указывающая на следующий этап, содержит данные, пережившие отбраковку. Обратите внимание: цифры здесь *весьма* приближительные.

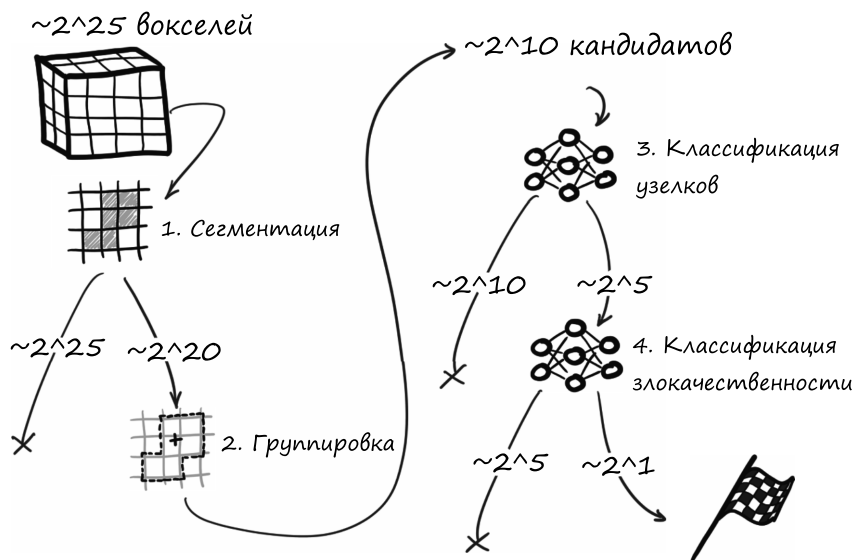


Рис. 14.4. Этапы нашего проекта сквозного обнаружения и приблизительный порядок объемов данных, отбрасываемых на каждом этапе

Рассмотрим приведенные на рис. 14.4 этапы более подробно.

1. *Сегментация.* Начинается с анализа целой КТ: сотни срезов или около 33 миллионов (2^{25}) вокселей (плюс-минус довольно много). Около 2^{20} вокселей помечаются как представляющие интерес, а это уже на несколько порядков меньше, чем на входе. Мы отбрасываем около 97 % вокселей (это 2^{25} слева, ведущие в X).
2. *Группировка.* Ничего не удаляет явным образом, однако уменьшает количество рассматриваемых элементов, поскольку мы объединяем воксели в узелки-кандидаты. Группировка производит около 1000 кандидатов (2^{10}) из 1 миллиона вокселей. Узелок размером $16 \times 16 \times 2$ вокселей содержит в общей сложности 2^{10} кандидатов¹.
3. *Классификация узелков.* На данном этапе отбрасывается большинство из оставшихся $\sim 2^{10}$ элементов. Из тысяч кандидатов на узелки у нас остались лишь десятки: около 25.
4. *Классификация злокачественности.* Наконец, классификатор злокачественности включает десятки узлов (2^5) и находит одно или два (2^1) раковых образования.

¹ Очевидно, что размер узелков бывает разный.

На каждом этапе проекта мы отбрасываем огромное количество данных, которые, по мнению модели, не имеют отношения к раковым образованиям. Мы перешли от миллионов точек данных к горстке опухолей.

ПОЛНОСТЬЮ АВТОМАТИЗИРОВАННЫЕ И ВСПОМОГАТЕЛЬНЫЕ СИСТЕМЫ

Существует разница между полностью автоматизированными системами и системами, которые должны дополнять способности человека. Автоматизированная система работает следующим образом: если часть данных помечается как нерелевантная, то исчезает навсегда. Но вообще, если мы создаем данные для дальнейшего анализа человеком, то должны дать ему возможность самому просматривать срезы и анализировать близкие промахи, а также аннотировать наши результаты с определенной степенью уверенности. Если бы мы разрабатывали систему для клинического использования, то нам нужно было бы тщательно продумать предполагаемое использование и убедиться, что дизайн системы хорошо поддерживает такую работу. Поскольку наш проект полностью автоматизирован, мы можем двигаться вперед, не задумываясь о том, как лучше выявлять возможные промахи и неуверенные ответы.

Теперь, когда мы нашли на изображении области, которые наша модель сегментации считает вероятными кандидатами, нам нужно вырезать этих кандидатов из КТ и передать их модулю классификации. К счастью, у нас уже есть список `candidateInfo_list` (см. предыдущий подраздел), поэтому нам нужно лишь сделать из него `DataSet`, положить его в `DataLoader` и перебрать данные. Столбец 1 прогнозов вероятности — прогнозируемая вероятность того, что перед нами узелок, и эту информацию нужно сохранить. Как и прежде, мы будем брать выходные данные из цикла (листинг 14.5).

Листинг 14.5. `nodule_analysis.py:357, .classifyCandidates`

```

        Опять же, мы получаем загрузчик данных и перебираем
        его, но на этот раз он содержит список кандидатов
def classifyCandidates(self, ct, candidateInfo_list):
    cls_dl = self.initClassificationDl(candidateInfo_list)
    classifications_list = []
    for batch_ndx, batch_tup in enumerate(cls_dl):
        input_t, _, _, series_list, center_list = batch_tup

        input_g = input_t.to(self.device)
        with torch.no_grad():
            _, probability_nodule_g = self.cls_model(input_g)
            if self.malignancy_model is not None:
                _, probability_mal_g = self.malignancy_model(input_g)
            else:
                probability_mal_g = torch.zeros_like(probability_nodule_g)

```

Отправка входных данных на устройство

Пропускаем данные через классификатор узелков

Если у нас есть модель злокачественных образований, ее тоже используем

```

zip_iter = zip(center_list,
               probability_nodule_g[:,1].tolist(),
               probability_mal_g[:,1].tolist())
for center_irc, prob_nodule, prob_mal in zip_iter:
    center_xyz = irc2xyz(center_irc,
                        direction_a=ct.direction_a,
                        origin_xyz=ct.origin_xyz,
                        vxSize_xyz=ct.vxSize_xyz,
                        )
    cls_tup = (prob_nodule, prob_mal, center_xyz, center_irc) ← Подсчет
                                                         результатов
    classifications_list.append(cls_tup)
return classifications_list

```

Отлично! Теперь мы можем применить к выходным вероятностям пороговое значение, чтобы получить список объектов, которые наша модель считает настоящими узелками.

В практических условиях мы, вероятно, передали бы данные на проверку рентгенологу. Опять же, пороговое значение можно скорректировать, чтобы увеличить количество ошибок в целях безопасности. Например, если бы пороговое значение было равно 0,3, а не 0,5, то на выходе оказалось бы больше кандидатов, которые не окажутся узелками, но снизится риск пропуска настоящих узелков (листинг 14.6).

Листинг 14.6. nodule_analysis.py:333, NoduleAnalysisApp.main

```

                                     Если тест run_validation не проходит,
                                     то выводим информацию...
if not self.cli_args.run_validation: ←
    print(f"found nodule candidates in {series_uid}:")
    for prob, prob_mal, center_xyz, center_irc in classifications_list:
        if prob > 0.5: ←
            s = f"nodule prob {prob:.3f}, "
            if self.malignancy_model:
                s += f"malignancy prob {prob_mal:.3f}, "
                s += f"center xyz {center_xyz}"
            print(s)
                                     ...по всем обнаруженным моделью
                                     сегментации кандидатам, которым
                                     классификатор назначил вероятность
                                     50 % и более

if series_uid in candidateInfo_dict: ←
    one_confusion = match_and_score(
        classifications_list, candidateInfo_dict[series_uid]
    )
    all_confusion += one_confusion
    print_confusion(
        series_uid, one_confusion, self.malignancy_model is not None
    )

print_confusion(
    "Total", all_confusion, self.malignancy_model is not None
)

```

Если у нас есть точные данные для проверки, то вычисляем и выводим матрицу ошибок и добавляем текущие результаты к общему количеству

Запустим этот сценарий для данной КТ из проверочного набора¹:

```

$ python3.6 -m p2ch14.nodule_analysis 1.3.6.1.4.1.14519.5.2.1.6279.6001
⇒ .592821488053137951302246128864
...
found nodule candidates in 1.3.6.1.4.1.14519.5.2.1.6279.6001.5928214880
⇒ 53137951302246128864:
nodule prob 0.533, malignancy prob 0.030, center xyz XyzTuple ←
⇒ (x=-128.857421875, y=-80.349609375, z=-31.300007820129395)
nodule prob 0.754, malignancy prob 0.446, center xyz XyzTuple
⇒ (x=-116.396484375, y=-168.142578125, z=-238.30000233650208)
...
nodule prob 0.974, malignancy prob 0.427, center xyz XyzTuple ←
⇒ (x=121.494140625, y=-45.798828125, z=-211.3000030517578)
nodule prob 0.700, malignancy prob 0.310, center xyz XyzTuple
⇒ (x=123.759765625, y=-44.666015625, z=-211.3000030517578)

```

Этот кандидат является злокачественным образованием с вероятностью 53 %, едва переходя порог вероятности 50 %. Классификация злокачественных новообразований присваивает очень низкую вероятность (3 %)

Узелок сочень высокой достоверностью и вероятностью злокачественности 42 %

Сценарий обнаружил всего 16 кандидатов в узелки. Поскольку мы используем проверочный набор, у нас есть полный набор аннотаций и информации о злокачественных новообразованиях для каждой КТ, с помощью которого мы можем создать матрицу ошибок. В строках — истинная информация (определенная аннотациями), а в столбцах результат, выданный нашей системой:

Прогноз: Complete Miss означает, что модель сегментации не обнаружила узелок, Filtered Out — работа классификатора, а Predicted Nodules — определенные системой узелки

ID скана	Complete Miss	Filtered Out	Pred. Nodule
1.3.6.1.4.1.14519.5.2.1.6279.6001.592821488053137951302246128864			
Non-Nodules		1088	15
Benign	1	0	0
Malignant	0	0	1

В строках приведена истинная информация

Столбец Complete Miss соответствует случаю, когда модель сегментации вообще не находит узелок. Поскольку она не помечает участки без узелков, мы оставляем данную ячейку пустой. Модель сегментации обучалась на высокое значение отклика, поэтому в ней много ложных узелков, но наш классификатор узелков хорошо справится с их отсеиванием.

Таким образом, на этом скане мы нашли один злокачественный узелок, но пропустили 17-й доброкачественный. Кроме того, через классификатор узлов прошло 15 ложноположительных результатов. Фильтрация по классификатору

¹ Мы выбрали эту КТ, поскольку в ней есть хорошее разнообразие результатов.

позволила многократно снизить количество ложноположительных результатов, которых было больше 1000! Как мы видели ранее, 1088 — это примерно $O(2^{10})$, что совпадает с нашими ожиданиями. Число 15 составляет около $O(2^4)$, что не далеко от $O(2^5)$, на которое мы ориентировались.

Прекрасно! Но какова общая картина?

14.4. КОЛИЧЕСТВЕННАЯ ОЦЕНКА

Теперь, когда у нас есть неофициальные свидетельства того, что с одной томографией модель справилась, оценим производительность нашей модели на всем проверочном наборе. Сделать это просто: мы прогоняем наш проверочный набор по предыдущему прогнозу и проверяем, сколько узелков получится, сколько будет пропущено и сколько кандидатов будет ошибочно идентифицировано как узелки.

Выполним приведенный ниже сценарий. При запуске на графическом процессоре это должно занять от получаса до часа. Выпив кофе (или даже вздремнув), вы получите результат:

```
$ python3 -m p2ch14.nodule_analysis --run-validation
```

```
...
Total
```

	Complete Miss	Filtered Out	Pred. Nodule
Non-Nodules		164893	2156
Benign	12	3	87
Malignant	1	6	45

Мы обнаружили 132 из 154 узелков, то есть 85 %. Из 22, которые мы пропустили, 13 отсеялись на этапе сегментации, поэтому именно отсюда стоит начать работу над улучшениями.

Около 95 % обнаруженных узелков — ложноположительные. Это, конечно, не здорово, но и не так критично. Врачу все равно проще просмотреть 20 кандидатов в поисках одного узелка, чем всю КТ. Мы рассмотрим данный вопрос более подробно в подразделе 14.7.2, но хотим подчеркнуть: вместо того, чтобы относиться к этим ошибкам как к «черному ящику», было бы неплохо исследовать ошибочно классифицированные случаи и поискать их общие черты. Есть ли у них характеристики, отличающие их от правильно классифицированных элементов? Можем ли мы найти способ использовать такие черты для улучшения работы модели?

А пока примем наши цифры как есть: неплохо, но и не идеально. В вашем случае точные числа могут получиться слегка другими. Ближе к концу этой главы мы предоставим несколько ссылок на документы и методы, которые могут помочь

вам улучшить результаты. Мы уверены, что с помощью вдохновения и экспериментов вы сможете добиться лучших результатов, чем показано здесь.

14.5. ПРОГНОЗИРОВАНИЕ ЗЛОКАЧЕСТВЕННОСТИ

Теперь, когда мы реализовали задачу обнаружения узелков от LUNA и можем делать собственные прогнозы, логично задать себе следующий вопрос: можем ли мы отличить злокачественные узлы от доброкачественных? Стоит сказать, что даже при наличии хорошей системы диагностика злокачественных новообразований, вероятно, требует более целостного взгляда на пациента, дополнительного, не связанного с КТ контекста и в конечном итоге биопсии, а не простого изучения отдельных узлов на КТ. Это не секундная задача, и врач будет заниматься ею в течение некоторого времени.

14.5.1. Получение информации о злокачественных новообразованиях

Задача LUNA посвящена обнаружению узелков и не содержит информации о злокачественных новообразованиях. В наборе данных LIDC-IDRI (<http://mng.bz/4A4R>) есть надмножество КТ-сканов, использованных для набора данных LUNA, а также информация о степени злокачественности выявленных опухолей. Удобно, что есть библиотека PyLIDC, которую можно легко установить с помощью команды:

```
$ pip3 install pylidc
```

Библиотека `pylidc` дает нам свободный доступ к дополнительной информации о злокачественных новообразованиях, которая нам и нужна. Как и в случае сопоставления аннотаций к кандидатам по местоположению, которое мы выполняли в главе 10, нам необходимо связать информацию аннотаций из LIDC с координатами кандидатов LUNA.

В аннотациях LIDC информация о злокачественных новообразованиях кодируется для каждого узелка и диагностирующего радиолога (до четырех вердиктов для одного и того же узелка) с помощью порядковой пятизначной шкалы от 1 (крайне маловероятно) до 5 (крайне подозрительно)¹. Эти аннотации сделаны лишь по изображению и зависят от предположений о пациенте. Чтобы преобразовать список чисел в одно логическое значение «да/нет», мы будем считать узлы злокачественными, если по крайней мере два радиолога оценили данный узел как «умеренно подозрительный» или выше. Обратите внимание:

¹ См. документацию по PyLIDC для получения полной информации: <http://mng.bz/Qyv6>.

этот критерий несколько произволен. На самом деле в литературе приводится множество способов обработки подобных данных, включая составление прогноза по этапам, использование средних значений или удаление из набора данных узелков, в отношении которых радиологи не сошлись во мнении.

Технически задача объединения данных будет такой же, как в главе 10, поэтому код здесь приводить не будем (зато он есть в репозитории кода для текущей главы). Воспользуемся расширенным CSV-файлом и составим набор данных, очень похожий на тот, который служил для классификатора узелков, с небольшим исключением: теперь мы обрабатываем только фактические узелки и определяем их злокачественность с помощью меток. Структурно эта задача очень похожа на балансировку, которую мы применили в главе 12, но вместо выборки из `pos_list` и `neg_list` мы берем из `mal_list` и `ben_list`. Как и в случае с классификатором узелков, мы хотим сохранить сбалансированность обучающих данных. Данные положим в класс `MalignancyLunaDataset`, который наследует от `LunaDataset`, но в целом очень похож на него.

Для удобства введем в сценарий `training.py` аргумент командной строки `dataset` и динамически используем класс набора данных, указанный в командной строке. В этом поможет функция `getattr`. Например, если `self.cli_args.dataset` — строка `MalignancyLunaDataset`, то берется тип `p2ch14.dsets.MalignancyLunaDataset` и присваивается `ds_cls`, как показано в листинге 14.7.

Листинг 14.7. `training.py:154, .initTrainDl`

```
ds_cls = getattr(p2ch14.dsets, self.cli_args.dataset)
train_ds = ds_cls(
    val_stride=10,
    isValSet_bool=False,
    ratio_int=1,
)
```

Динамический просмотр имени класса

Напомним, что мы выполняем балансировку обучающих данных так, чтобы сведений о злокачественных и доброкачественных новообразованиях было поровну

14.5.2. Базовый уровень для вычисления площади под кривой: классификация по диаметру

Всегда полезно иметь базовый уровень, чтобы оценить, какая производительность лучше, чем ничего. Мы могли бы выбрать более подходящий критерий, но в данном случае в качестве предиктора злокачественности можем использовать диаметр — более крупные узлы с большей вероятностью будут злокачественными. На этапе 2Б на рис. 14.5 показана новая метрика, с помощью которой мы можем сравнить классификаторы.

Диаметр узелка вообще мог выступать в качестве единственного входа в гипотетический классификатор, предсказывающий злокачественность узелка. Это был бы не очень хороший классификатор, но, оказывается, утверждение «Все,

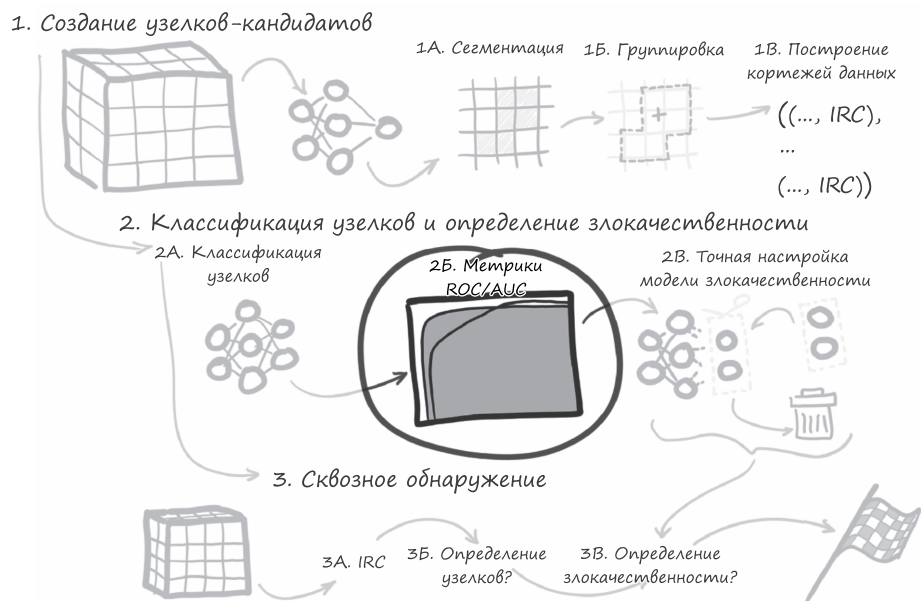


Рис. 14.5. План этой главы. Выделен график метрики площади под кривой

что превышает этот порог X , является злокачественным» лучше прогнозирует злокачественность, чем мы могли бы ожидать. Разумеется, ключевое значение имеет выбор правильного порогового значения. Легко сказать, что все крупные образования — злокачественные, а мелкие — нет, но никто не отменяет средних значений, где бывают и те и другие.

Как мы помним из главы 12, количество истинно положительных, ложноположительных, истинно отрицательных и ложноотрицательных результатов меняется в зависимости от того, какое пороговое значение мы выбираем. Уменьшив пороговое значение, выше которого считается, что узелок является злокачественным, мы увеличим количество истинно положительных результатов, но также и количество ложноположительных результатов. *Доля ложноположительных результатов* (false positive rate, FPR) равна $FP / (FP + TN)$, а *доля истинно положительных* (true positive rate, TPR) равна $TP / (TP + FN)$. Это вы тоже, вероятно, помните из главы 12.

Зададим диапазон пороговых значений. Нижняя граница — это значение, при котором все образцы классифицируются как положительные, а верхняя граница, напротив, определяет, когда образцы классифицируются как отрицательные. На одном конце значения FPR и TPR будут равны 0, поскольку положительных результатов не будет вообще, а на другом — равны 1, так как TN и FN не будет (все результаты положительные).

**РЕЦЕПТ ИЗМЕРЕНИЯ ЛОЖНОПОЛОЖИТЕЛЬНЫХ РЕЗУЛЬТАТОВ: ТОЧНОСТЬ
ИЛИ ОТНОШЕНИЯ ЛОЖНОПОЛОЖИТЕЛЬНЫХ**

Показатель FPR и точность из главы 12 — это коэффициенты (от 0 до 1), которые измеряют довольно близкие вещи. Как мы уже обсуждали, точность равна $TP / (TP + FP)$ и показывает, сколько элементов данных, классифицированных как положительные, являются таковыми на самом деле. FPR — это $FP / (FP + TN)$, то есть количество отрицательных элементов, классифицированных как положительные. Для сильно несбалансированных наборов данных (как в задаче классификации узелков) наша модель может дать хороший показатель FPR (который тесно связан перекрестной энтропией), а точность и метрика F1 могут оказаться плохими. Низкий показатель FPR означает, что мы отсеиваем много того, что нас не интересует, но в нашей задаче и впрямь слишком много сена вокруг иголок.

Данные о размерах узелков колеблются от 3,25 мм (самый маленький) до 22,78 мм (самый большой). Выбрав пороговое значение где-то между этими двумя значениями, мы сможем вычислить FPR и TPR в зависимости от него. Если условимся, что FPR — это X , а TPR — Y , то можем нанести на график точку, соответствующую пороговому значению, и построить график зависимости FPR от TPR для каждого возможного порогового значения. В результате получится кривая ROC, или *рабочая характеристика приемника*, показанная на рис. 14.6. Заштрихованная область — это *площадь под кривой* (receiver operating characteristic, ROC), или AUC (area under the (ROC) curve). Ее значение находится в диапазоне от 0 до 1, и чем оно больше, тем лучше¹.

Здесь мы возьмем два конкретных пороговых значения: диаметры 5,42 мм и 10,55 мм. Мы выбрали именно их, поскольку из них получаются разумные конечные точки для диапазона пороговых значений, из которого можно выбрать нужное. Пороговое значение меньше 5,42 мм негативно скажется на TPR. Значения больше 10,55 мм приводят к отметке злокачественных узлов как доброкачественных. Оптимальное пороговое значение для этого классификатора, вероятно, будет где-то посередине.

Как вообще вычислены приведенные на графике значения? Сначала мы получаем список с информацией о кандидатах, отфильтровываем аннотированные узелки, получаем метку злокачественности и диаметр. Для удобства мы также получаем количество доброкачественных и злокачественных узлов (листинг 14.8).

¹ Обратите внимание, что случайные прогнозы на сбалансированном наборе данных дают значение $AUC = 0,5$, так что наш классификатор должен по крайней мере превосходить это значение.

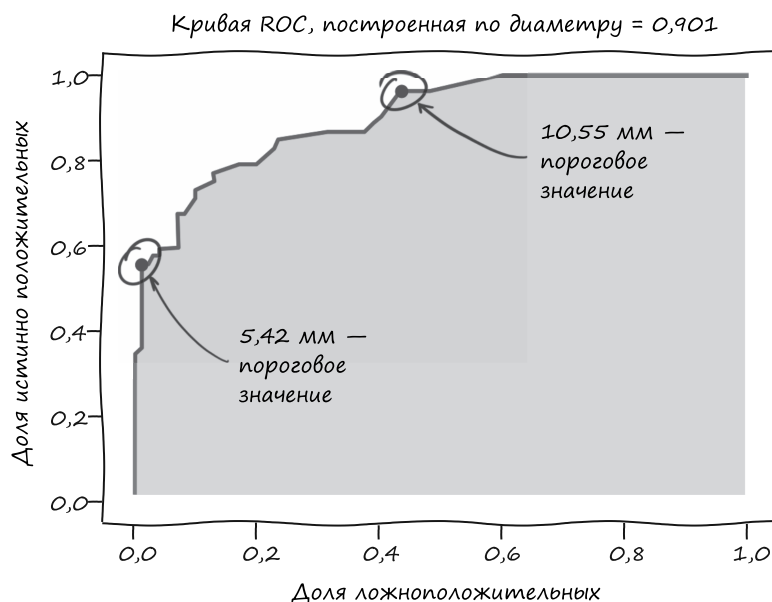


Рис. 14.6. Кривая ROC, построенная по диаметру узелка

Листинг 14.8. p2ch14_malben_baseline.ipynb

```
# In[2]:
ds = p2ch14.dsets.MalignantLunaDataset(val_stride=10, isValSet_bool=True)
nodules = ds.ben_list + ds.mal_list
is_mal = torch.tensor([n.isMal_bool for n in nodules])
diam = torch.tensor([n.diameter_mm for n in nodules])
num_mal = is_mal.sum()
num_ben = len(is_mal) - num_mal
```

Берем обычный набор данных, а именно список доброкачественных и злокачественных узелков

Получаем списки с информацией о злокачественности и диаметром

Для нормализации TPR и FPR берем количество злокачественных и доброкачественных узелков

Чтобы вычислить кривую ROC, нам нужен массив возможных пороговых значений. Его мы берем из функции `torch.linspace`, которая принимает два граничных элемента. Мы хотим начать с нулевого количества положительных прогнозов, поэтому идем от максимального порога к минимальному, то есть от 3,25 до 22,78:

```
# In[3]:
threshold = torch.linspace(diam.max(), diam.min())
```

Затем мы строим двумерный тензор, в котором строки соответствуют пороговому значению, столбцы содержат информацию для каждого элемента данных, а значением выступает признак положительности этого элемента. Затем полученный тензор типа `Boolean` фильтруется по тому, является ли метка элемента

злокачественной или доброкачественной. Затем мы суммируем строки, чтобы подсчитать количество значений True. Делим это количество на злокачественные или доброкачественные узелки и получаем значения TPR и FPR — две координаты кривой ROC:

```
# In[4]:
predictions = (diam[None] >= threshold[:, None])
tp_diam = (predictions & is_mal[None]).sum(1).float() / num_mal
fp_diam = (predictions & ~is_mal[None]).sum(1).float() / num_ben
```

Индексация по None добавляет измерение размера 1, как метод .unsqueeze(nsx). Это дает нам двумерный тензор с информацией о злокачественности узелков при заданном диаметре

С помощью матрицы прогнозов мы можем вычислить TPR и FPR для каждого диаметра, суммируя столбцы

Чтобы вычислить площадь под кривой, мы используем численное интегрирование по правилу трапеций (https://en.wikipedia.org/wiki/Trapezoidal_rule), где мы умножаем среднее значение TPR (по оси Y) между двумя точками на разность двух FPR (по оси X) — площади трапеций между двумя точками графика. Затем суммируем площади трапеций:

```
# In[5]:
fp_diam_diff = fp_diam[1:] - fp_diam[:-1]
tp_diam_avg = (tp_diam[1:] + tp_diam[:-1])/2
auc_diam = (fp_diam_diff * tp_diam_avg).sum()
```

Запускаем функцию `pyplot.plot(fp_diam, tp_diam, label=f"diameter baseline, AUC={auc_diam:.3f}")` (вместе с соответствующей настройкой графика в ячейке 8) и получаем график, который был приведен на рис. 14.6.

14.5.3. Повторное использование весов: тонкая настройка

Один из способов быстро получить результаты (а зачастую и обойтись гораздо меньшим объемом данных) — не задавать веса случайно, а взять сеть, обученную какой-либо задаче со связанными данными. Это называется *передачей обучения* или, когда речь идет об обучении нескольких последних слоев, *тонкой настройкой*. На выделенной части на рис. 14.7 видно, что на этапе 2В мы собираемся вырезать последнюю часть модели и заменить ее чем-то новым.

В главе 8 говорилось, что промежуточные значения можно интерпретировать как признаки, извлеченные из изображения. В качестве признаков могут выступать края, углы или какой-то заметный для модели шаблон. До глубокого обучения очень часто использовались созданные вручную признаки, подобные тем, с которыми мы экспериментировали, когда начали работу со свертками. Благодаря глубокому обучению сеть извлекает из данных признаки, полезные для поставленной задачи, например, различения классов. Теперь у нас есть тонкая настройка — смесь древних (им аж *десять* лет!) методов применения

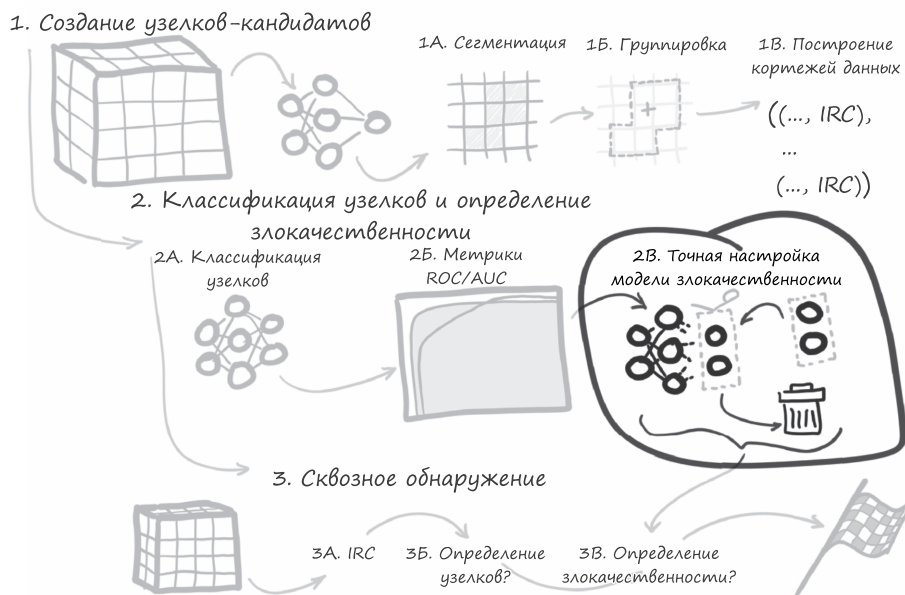


Рис. 14.7. План этой главы. Выделен этап тонкой настройки

заготовленных признаков и нового метода использования изученных признаков. Мы рассматриваем некоторую (часто большую) часть сети как некий *распознаватель признаков*, а обучаем только относительно небольшую часть поверх нее.

Обычно это прекрасно работает. Предварительно обученные на ImageNet сети, как мы видели в главе 2, очень полезны в качестве экстракторов признаков во многих задачах, связанных с естественными изображениями, — иногда они также прекрасно работают для совершенно разных входных данных, от картин или их стилизованных имитаций до аудиоспектрограмм. Есть случаи, когда эта стратегия работает хуже. Например, одна из распространенных стратегий дополнения данных в обучающих моделях на ImageNet — это случайное переворачивание изображений: собака, смотрящая вправо, относится к тому же классу, что и собака, смотрящая влево. Обнаруженные на перевернутых изображениях признаки окажутся очень похожи. Но если мы сейчас попытаемся использовать предварительно обученную модель для задачи, где есть разница между левым и правым, то, скорее всего, у нас возникнут проблемы с точностью. Если сеть должна идентифицировать дорожные знаки, то знак «Поверните налево» будет сильно отличаться от «Поверните направо» и обученная на ImageNet сеть, вероятно, будет очень часто ошибаться¹.

¹ Попробуйте сделать это сами на наборе данных German Traffic Sign Recognition Benchmark, на <http://mng.bz/XPZ9>.

В нашем случае у нас есть сеть, обученная на аналогичных данных: сеть классификации узелков. Попробуем использовать ее.

Ради наглядности сохраним простой подход к тонкой настройке. В архитектуре модели на рис. 14.8 выделены два представляющих интерес момента: последний сверточный блок и модуль `head_linear`. Самая простая тонкая настройка — это удаление `head_linear`, то есть просто сохранение случайной инициализации. Попробовав данный вариант, рассмотрим другой: переобучим `head_linear` и последний сверточный блок.

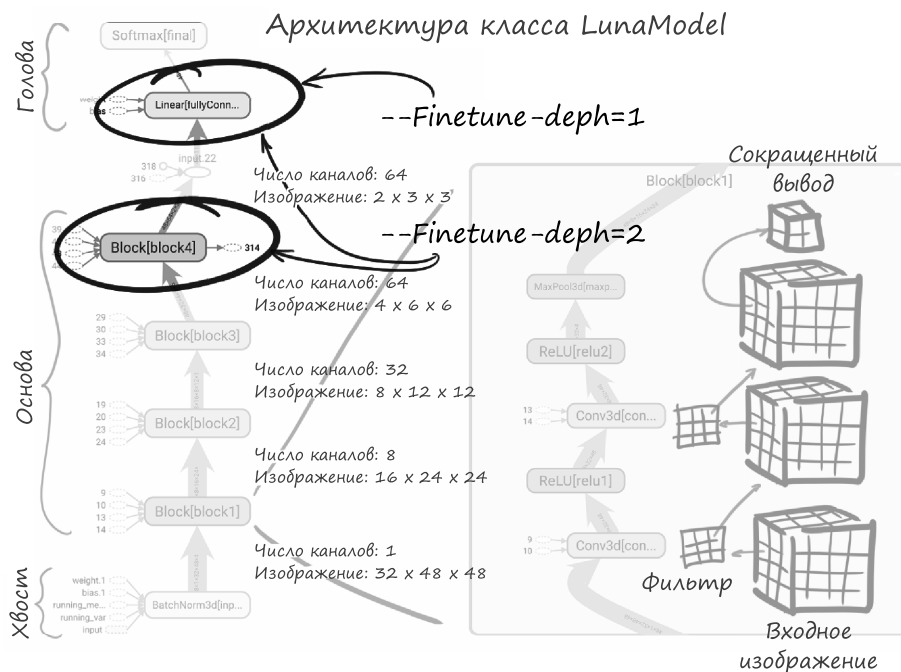


Рис. 14.8. Архитектура модели из главы 11 с выделенными весами глубины 1 и глубины 2

Нам нужно сделать следующее:

- загрузить нужные начальные веса модели, за исключением последнего линейного слоя, в котором нужна случайная инициализация;
- отключить градиенты для параметров, которые мы не хотим обучать (все, кроме параметров с именами, начинающимися со слова `head`).

Когда мы выполняем тонкую настройку обучения на чем-то, помимо `head_linear`, сам `head_linear` инициализируется случайно, поскольку мы считаем, что

предыдущие слои извлечения признаков могут неидеально подходить для нашей задачи, но по крайней мере зададут разумную отправную точку. Это легко: мы добавляем код загрузки в настройку модели (листинг 14.9).

Листинг 14.9. training.py:124, .initModel

```

Отфильтровывает модули верхнего уровня, у которых
есть параметры (в отличие от функции активации)
d = torch.load(self.cli_args.finetune, map_location='cpu')
model_blocks = [
    n for n, subm in model.named_children()
    if len(list(subm.parameters())) > 0
]
finetune_blocks = model_blocks[-self.cli_args.finetune_depth:]
model.load_state_dict(
    {
        k: v for k, v in d['model_state'].items()
        if k.split('.')[0] not in model_blocks[-1]
    },
    strict=False,
)
for n, p in model.named_parameters():
    if n.split('.')[0] not in finetune_blocks:
        p.requires_grad_(False)

```

Берет последние finetune_depth блоков. По умолчанию (при тонкой настройке) один блок

Отфильтровывает последний блок (последнюю линейную часть) и не загружает его. Работая с полностью инициализированной моделью, мы бы сперва классифицировали почти все узелки как злокачественные, поскольку этот вывод означает «узелок» в классификаторе, с которого мы начинаем

Передача strict=False позволяет нам загружать только некоторые веса модуля (без отфильтрованных)

Для всех, кроме finetune_blocks, градиенты не нужны

Готово! Запустим обучение, выполнив команду:

```

python3 -m p2ch14.training \
    --malignant \
    --dataset MalignantLunaDataset \
    --finetune data/part2/models/cls_2020-02-06_14.16.55_final-nodule-
    nonmodule.best.state \
    --epochs 40 \
    malben-finetune

```

Запустим модель на проверочном наборе и получим кривую ROC, показанную на рис. 14.9. Результат намного лучше, чем при случайной инициализации, но мы все равно не перешли за базовый уровень, поэтому нужно понять, в чем проблема.

На рис. 14.10 показаны графики TensorBoard, построенные по результатам обучения. Глядя на потери при проверке, мы видим, что AUC медленно увеличивается, а потери уменьшаются, но потери при обучении, кажется, выходят на довольно высокое плато (скажем, 0,3), а не стремятся к нулю. Мы могли бы запустить более длительное обучение, чтобы проверить, не слишком ли оно медленное. Однако, сравнивая результат с прогрессией потерь, о которой мы говорили в главе 5 на рис. 5.14, мы видим, что величина потерь не выравнивается так сильно, как в случае А на рисунке, но в целом имеет место такая же стагнация потерь. Тогда случай А указывал на то, что у нас недостаточно мощностей, поэтому нужно рассмотреть следующие три возможные причины:

- признаки (результаты последней свертки), полученные путем обучения сети классификации узлов, бесполезны для обнаружения злокачественных новообразований;
- у головы (а мы обучаем только ее) недостает мощности;
- общая мощность сети может быть слишком мала.

Если обучения только полносвязной части в методе тонкой настройки недостаточно, то нужно попробовать включить в обучение последний сверточный блок. К счастью, мы ввели для этого параметр, так что сделать это нетрудно:

```
python3 -m p2ch14.training \
    --malignant \
    --dataset MalignantLunaDataset \
    --finetune data/part2/models/cls_2020-02-06_14.16.55_final-nodule-
    nonmodule.best.state \
    --finetune-depth 2 \
    --epochs 10 \
    malben-finetune-twolayer
```

← Это новый параметр командной строки

После этого мы можем сравнить новую лучшую модель с базовым уровнем. Рисунок 14.11 выглядит уже лучше! Около 75 % злокачественных узлов отмечены правильно почти без ложноположительных результатов. Это явно лучше, чем 65 %, которые дает базовый уровень. При попытке выйти за пределы 75 % производительность нашей модели возвращается к базовому уровню. Когда мы вернемся к проблеме классификации, нужно будет выбрать точку на кривой ROC, чтобы сбалансировать истинно положительные и ложноположительные результаты.

Мы достигли примерно того же качества, что и на базовом уровне, и на этом остановимся. В разделе 14.7 будет приведено множество вещей, которые вы можете изучить, чтобы улучшить результаты, но в данной книге мы не будем их рассматривать.

Из кривых потерь на рис. 14.12 видно, что модель очень рано достигает переобучения. Таким образом, дальше нужно попробовать еще какие-нибудь методы регулирования ее работы. Но этим вы займетесь сами.

Существуют и более изощренные методы тонкой настройки. Некоторые любят размораживать слои один за другим, начиная сверху. Другие предлагают обучать более поздние слои с обычной скоростью, а нижние — медленнее. PyTorch поддерживает использование различных параметров оптимизации, таких как скорость обучения, затухание веса и импульс, путем разделения *параметров на группы*. Получаются списки параметров с отдельными гиперпараметрами (<https://pytorch.org/docs/stable/optim.html#per-parameter-options>).

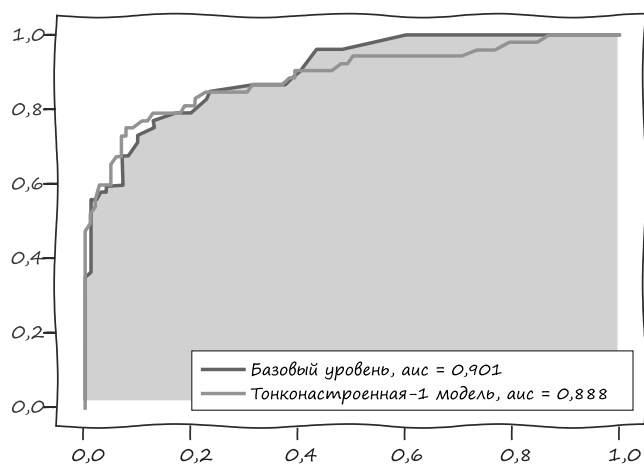


Рис. 14.9. ROC-кривая доработанной модели с переобученным конечным линейным слоем. Не так уж плоха, но и не дотягивает до базового уровня

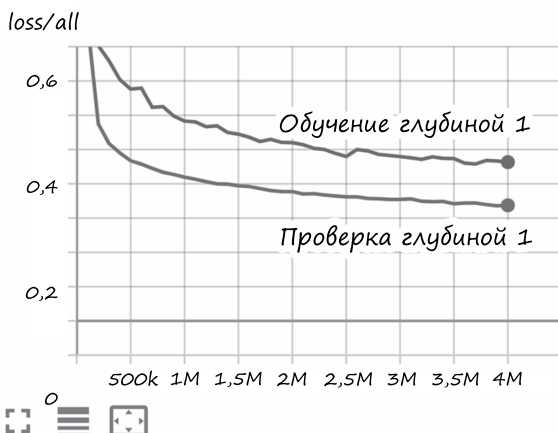
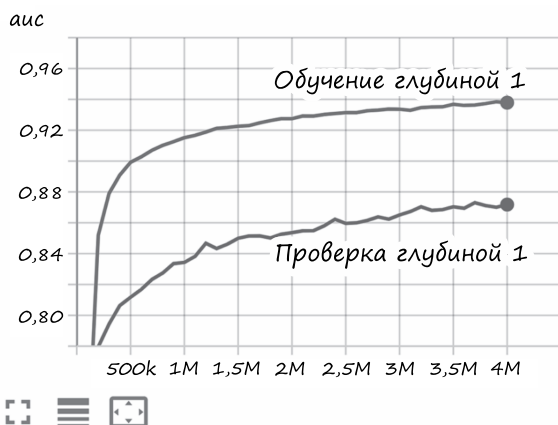


Рис. 14.10. Кривая AUC (вверху) и потери (внизу) для случая тонкой настройки последнего линейного слоя

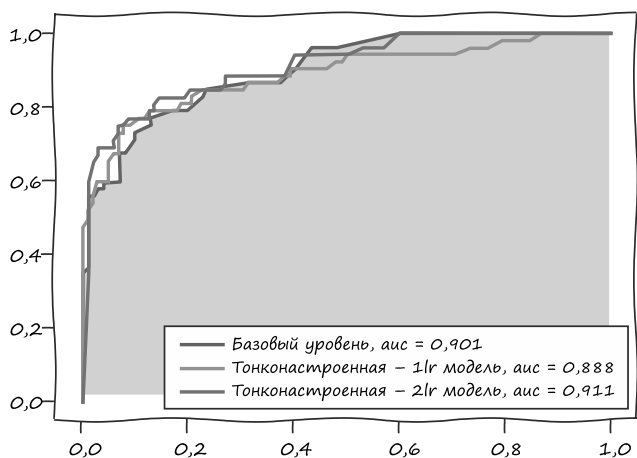


Рис. 14.11. ROC-кривая модифицированной модели. Теперь мы очень близки к базовому уровню

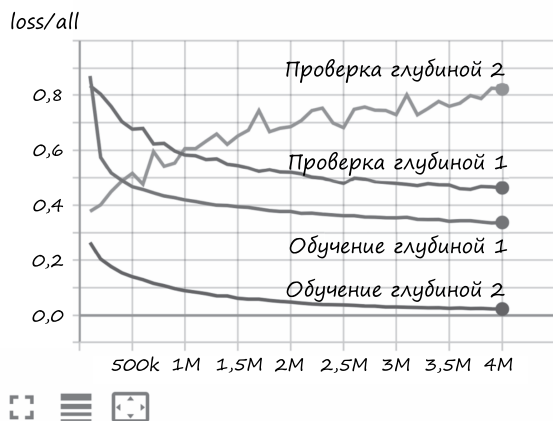
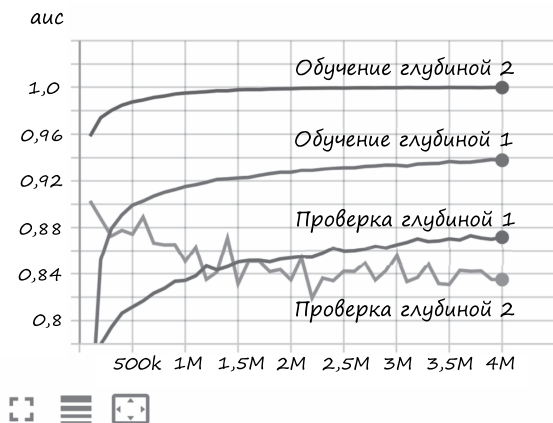


Рис. 14.12. Кривая AUC (вверху) и потери (внизу) для тонкой настройки последнего сверточного блока и полносвязного слоя

14.5.4. Больше данных в TensorBoard

Пока модель обучается заново, возможно, стоит добавить в TensorBoard еще кое-какие выходные данные, которые помогут понять, как у нас дела. Для гистограмм в TensorBoard есть готовая функция записи. Для ROC такой функции нет, а это значит, что мы можем познакомиться с интерфейсом Matplotlib.

Гистограммы

Мы можем взять значения вероятностей злокачественности и построить из них гистограмму. Даже две: одну для (по имеющимся данным) доброкачественных и одну для злокачественных узлов. Эти гистограммы дают нам детальное представление о выходных данных модели и показывают, существуют ли большие наборы выходных вероятностей, определенных полностью неправильно.

ПРИМЕЧАНИЕ

В целом отображение данных в правильной форме крайне важно для извлечения из них информации. Если у вас есть много чрезвычайно надежных и правильных классификаций, то вы можете исключить самую крайнюю часть слева. Вывод нужной информации на экран обычно требует тщательного обдумывания и различных подходов. Экспериментируйте с выходными данными, но также помните о том, что меняете определение метрики без изменения имени. Если вы не будете соблюдать порядок в схемах именования или удалении уже недействительных наборов данных, может оказаться, что вы сравниваете яблоки с апельсинами.

Сначала выделим место в тензоре `metrics_t`, который хранит данные (листинг 14.10). Напомним, что мы определили индексы где-то рядом с вершиной.

Листинг 14.10. training.py:31

```
METRICS_LABEL_NDX=0
METRICS_PRED_NDX=1
METRICS_PRED_P_NDX=2
METRICS_LOSS_NDX=3
METRICS_SIZE = 4
```

← Новый индекс, в котором хранятся вероятности предсказаний
(а не предсказания до применения порогового значения)

Закончив с этим, мы можем вызвать метод `writer.add_histogram` с меткой, данными и счетчиком `global_step`, равным количеству представленных обучающих элементов данных. Это похоже на скалярный вызов, выполненный ранее. Мы также передаем `bins` в фиксированном масштабе (листинг 14.11).

Листинг 14.11. training.py:496, .logMetrics

```
bins = np.linspace(0, 1)

writer.add_histogram(
    'label_neg',
    metrics_t[METRICS_PRED_P_NDX, negLabel_mask],
    self.totalTrainingSamples_count,
    bins=bins
```

```

)
writer.add_histogram(
    'label_pos',
    metrics_t[METRICS_PRED_P_NDX, posLabel_mask],
    self.totalTrainingSamples_count,
    bins=bins
)

```

Теперь мы можем взглянуть на распределение прогнозов для доброкачественных элементов и на их изменение по мере прохождения эпох. Рассмотрим основные особенности гистограмм на рис. 14.13. Как и следовало ожидать, если наша сеть что-то изучает, то в верхнем ряду доброкачественных образцов и не узелков есть гора элементов, в злокачественности которых сеть не уверена. Такая же гора злокачественных элементов имеется справа.

Взглянув повнимательнее, мы увидим проблему мощности, связанную с тонкой настройкой лишь одного слоя. Из серии гистограмм в верхнем левом углу видно, что масса слева несколько рассредоточена и не сильно уменьшается.

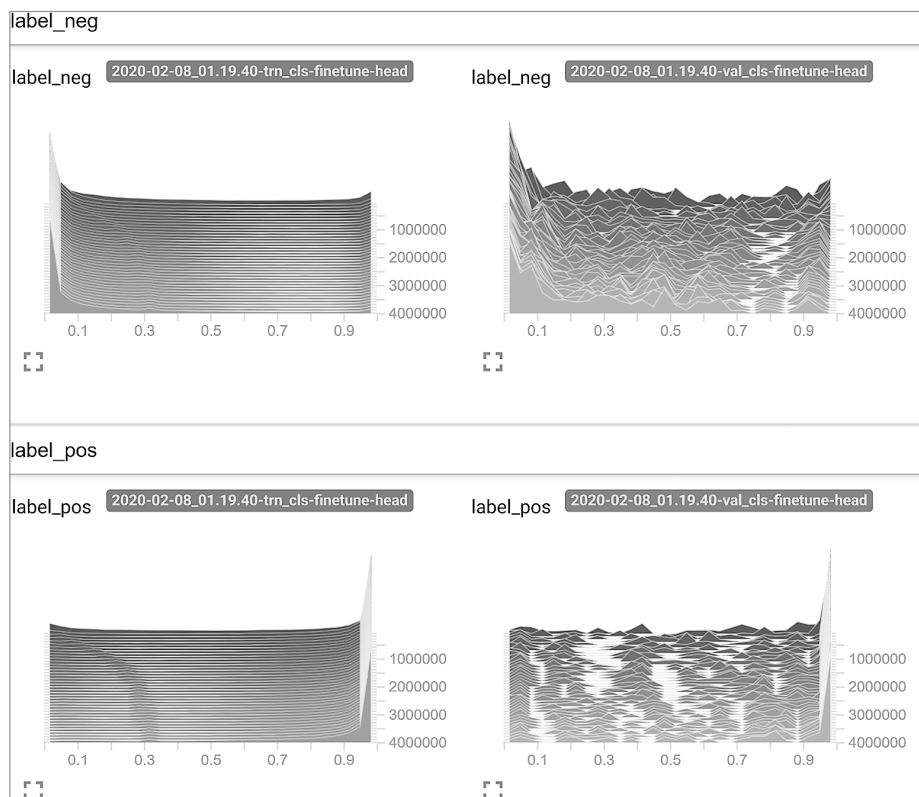


Рис. 14.13. Гистограммы TensorBoard для случая точной настройки только головы

Есть даже небольшой пик около 1,0, и довольно много вероятностной массы рассредоточено по всему диапазону. Это отражает значение потерь, которое не хочет опускаться ниже 0,3.

Даже этой информации о потерях при обучении достаточно, чтобы не смотреть дальше, но все же продолжим. В результатах проверки с правой стороны видно, что масса вероятности вдали от «правильной» стороны больше у незлокачественных образцов на верхней правой диаграмме, чем у злокачественных на нижней правой диаграмме. Таким образом, сеть чаще ошибается в незлокачественных образцах, чем в злокачественных. Это может заставить нас задуматься о перебалансировке данных, чтобы сеть получала больше доброкачественных образцов. Но опять же, это когда мы делаем вид, что с обучением слева все в порядке. Обычно, впрочем, следует сначала исправлять именно обучение!

Для сравнения взглянем на тот же график для тонкой настройки глубины 2 (рис. 14.14). В процессе обучения (две левые диаграммы) появились очень резкие пики при правильном ответе. Это говорит о том, что обучение работает хорошо.

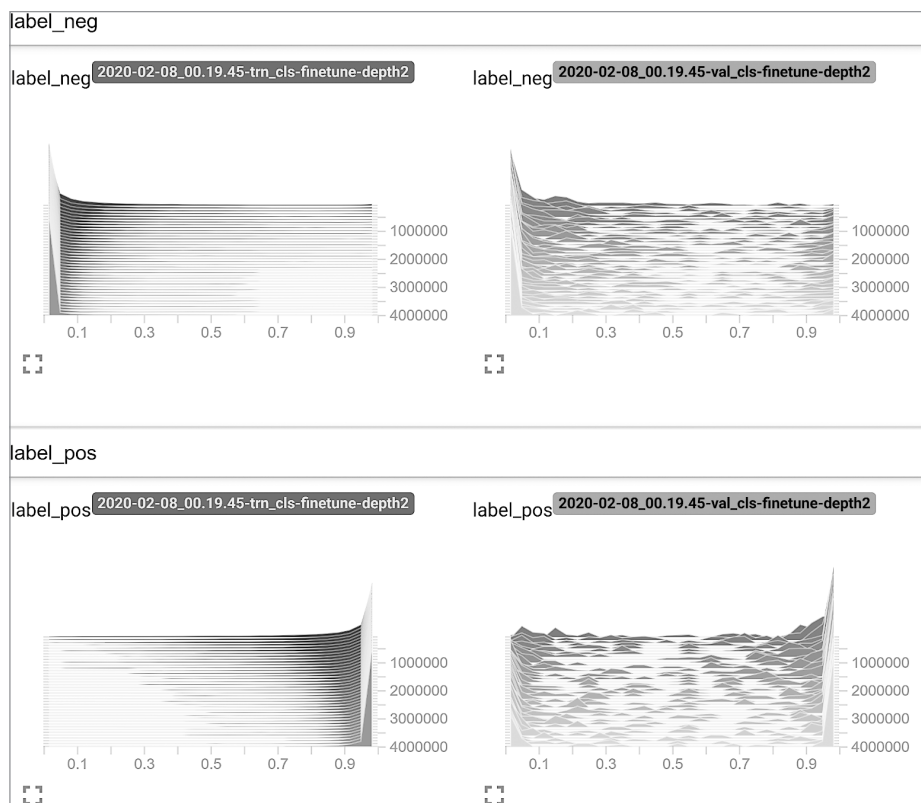


Рис. 14.14. Отображение гистограммы TensorBoard для тонкой настройки глубины 2

Что касается проверки, то теперь мы видим, что наиболее выраженный артефакт — небольшой пик при 0 прогнозируемой вероятности злокачественного новообразования на гистограмме в правом нижнем углу. Итак, наша систематическая проблема в том, что мы ошибочно классифицируем злокачественные образцы как незлокачественные. (Это противоположно тому, что было раньше!) Возникает переобучение, которое мы видели в случае двухслойной точной настройки. Наверное, было бы неплохо создать несколько подобных изображений с целью увидеть, что происходит.

ROC и другие кривые в TensorBoard

Как упоминалось ранее, TensorBoard изначально не поддерживает рисование кривых ROC. Зато мы можем экспортировать любой график из Matplotlib. Подготовка данных выглядит так же, как в подразделе 14.5.2: мы используем данные, которые тоже нанесли на гистограмму, для вычисления TPR и FPR — `tpf` и `fpr` соответственно. Мы снова наносим данные на график, однако на этот раз отслеживаем `pyplot.figure` и передаем ее в метод `SummaryWrite.add_figure` (листинг 14.12).

Листинг 14.12. training.py:482, .logMetrics

<p>Создаем график Matplotlib. Обычно этого не нужно делать, поскольку это неявно делается в Matplotlib, но в данном случае это необходимо</p>	<p>Используем произвольные функции pyplot</p>	<p>Добавляет график в TensorBoard</p>
<pre> fig = pyplot.figure() pyplot.plot(fpr, tpr) writer.add_figure('roc', fig, self.totalTrainingSamples_count) </pre>		

Поскольку график передается в TensorBoard как изображение, оно отображается под этим заголовком. Мы не рисовали кривую сравнения или другие кривые, чтобы не загромождать код вызова функции, но вообще в этом месте можно было использовать любые возможности Matplotlib. На рис. 14.15 мы снова видим, что точная настройка глубины 2 (*слева*) дает переобучение, а точная настройка только головы (*справа*) — нет.

14.6. КАКОВ ДИАГНОЗ?

В соответствии с этапами 3А, 3Б и 3В на рис. 14.16 теперь нужно запустить полный конвейер от сегментации на этапе 3А, *слева*, до модели злокачественности на этапе 3В, *справа*. Облегчает дело то, что почти весь код уже написан! Нам нужно лишь сшить все воедино: настал момент написания и запуска сквозного сценария для постановки диагноза.

Первые намеки на работу с моделью злокачественности появлялись в коде еще в подразделе 14.3.3. Если мы передаем аргумент `--malignancy-path` сценарию

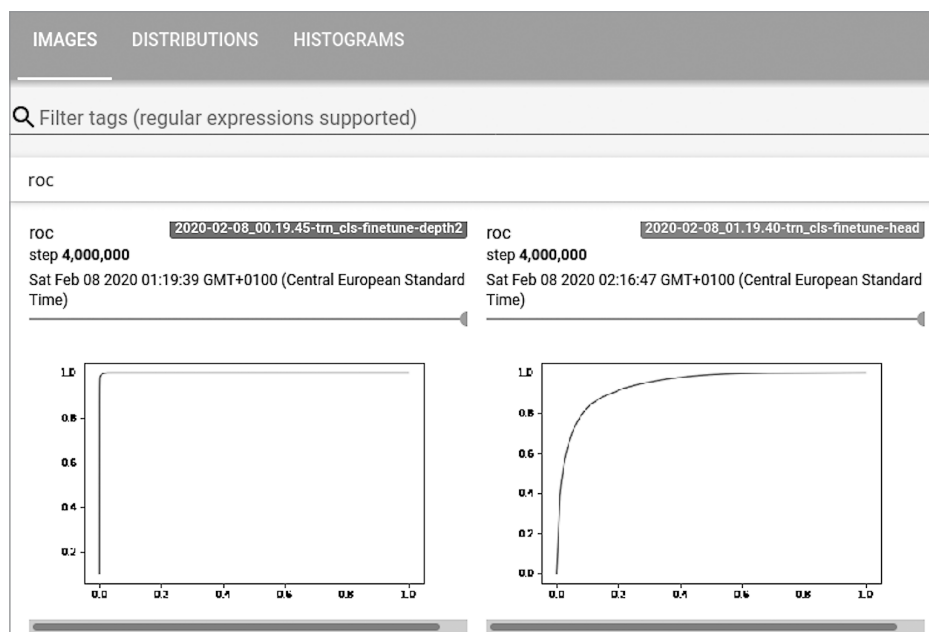
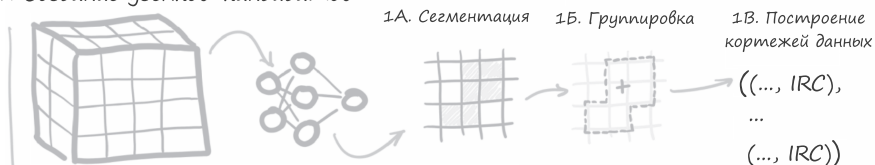
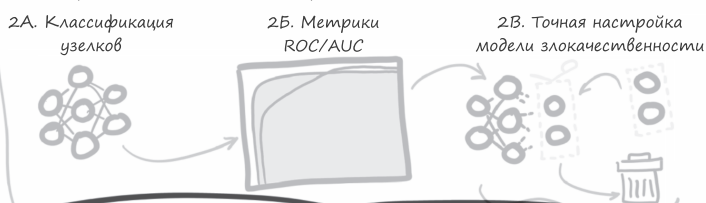


Рис. 14.15. Обучение ROC-кривых в TensorBoard. Ползунок позволяет просматривать разные итерации

1. Создание узелков-кандидатов



2. Классификация узелков и определение злокачественности



3. Сквозное обнаружение



Рис. 14.16. План этой главы. Выделен этап сквозного обнаружения

`nodule_analysis`, то он запускает модель злокачественности, найденную по переданному пути, и выводит информацию. Это работает как для одиночного сканирования, так и для опции `--run-validation`.

Имейте в виду, что сценарий, вероятно, будет выполняться некоторое время. Анализ даже 89 КТ в проверочном наборе занял около 25 минут¹.

Посмотрим, что у нас получится:

Total		Complete Miss	Filtered Out	Pred. Benign	Pred. Malignant
Non-Nodules			164893	1593	563
Benign		12	3	70	17
Malignant		1	6	9	36

Не так уж плохо! В сквозном процессе мы выявляем около 85 % узелков и правильно помечаем около 70 % злокачественных². У нас получается много ложноположительных результатов, но даже 16 ложноположительных на каждый истинный уже снижает количество необходимых исследований (ну, если бы не 30 % ложноотрицательных результатов). В главе 9 мы упоминали, что это не тот уровень, на котором можно было бы заработать миллионы стартапа в области медицинского ИИ³, но вполне неплохая отправная точка. В целом мы должны быть очень довольны получением осмысленных результатов, тем более что *настоящая* цель состояла в изучении глубокого обучения.

Можно было бы более внимательно исследовать узелки, классифицированные неправильно. Но тут стоит вспомнить, что в нашей задаче даже радиологи, которые составляли аннотации, разошлись во мнениях. Мы могли бы стратифицировать проверочный набор по тому, насколько точно узел определяется как злокачественный.

14.6.1. Наборы для обучения, проверки и тестирования

Есть одно предостережение, о котором мы должны упомянуть. Мы не обучали нашу модель на проверочном наборе явным образом, *выбор* количества эпох для обучения зависел от производительности модели на проверочном наборе. Это

¹ Большая часть временных задержек связана с обработкой компонентов SciPy. На момент написания нам неизвестно, есть ли более быстрая реализация.

² Напомним, что ранее показатель ROC «75 % почти без ложных срабатываний» рассматривал классификацию злокачественных новообразований изолированно. Здесь мы отфильтровываем семь злокачественных узлов еще до того, как дойдем до классификатора злокачественности.

³ Будь это так, мы бы это и сделали, а не писали бы книгу!

небольшие утечки данных. Следует ожидать небольшого ухудшения реальной производительности, поскольку маловероятно, что любая модель, которая лучше всего работает на нашем проверочном наборе, будет одинаково хорошо работать на любом другом наборе данных (по крайней мере в среднем).

Поэтому специалисты-практики часто разбивают данные на *три* набора:

- *обучающий*, такой же, как у нас;
- *проверочный*, используется для определения того, какую эпоху эволюции модели считать лучшей;
- *тестовый*, служит для фактического прогнозирования производительности модели (выбранной проверочным набором) на невидимых реальных данных.

Добавление третьего набора вынуждает нас извлекать из обучающих данных еще один фрагмент, что было бы несколько болезненно, учитывая, как сложно нам было бороться с переобучением. Вдобавок это усложнило бы задачу представления данных, так что мы намеренно не использовали такой подход. Если бы у нас был проект, в котором можно было бы получить больше данных, а цель состояла в создании наилучшей системы для реальной работы, то нам пришлось бы принять другое решение и искать больше данных для использования в качестве независимого тестового набора.

Вывод здесь заключается в том, что модели способны приобретать неявные предубеждения. Мы должны с особой осторожностью контролировать утечки информации на каждом этапе и проверять их отсутствие на независимых данных, насколько это возможно. Цена за использование кратчайших путей непосильной ношей сваливается на нас на более позднем этапе и в самый неподходящий момент: когда мы приближаемся к производству.

14.7. ЧТО ДАЛЬШЕ? ДОПОЛНИТЕЛЬНЫЕ ИСТОЧНИКИ ВДОХНОВЕНИЯ (И ДАННЫХ)

Дальнейшие улучшения на данном этапе оценить трудно. Проверочный набор для модели классификации содержит 154 узелка, и наша модель классификации узелков обычно правильно определяет как минимум 150 из них, при этом большая часть отклонений возникает из-за изменений в обучении от эпохи к эпохе. Даже если бы мы значительно усовершенствовали модель, то не достигли бы достаточной точности в проверочном наборе, чтобы быть абсолютно уверенными во влиянии данного усовершенствования на что-то! Это очень заметно и в классификации доброкачественных и злокачественных новообразований, где потери во время проверки выглядят в значительной степени зигзагообразно. Уменьши

мы шаг проверки с 10 до 5, размер нашего проверочного набора удвоился бы за счет одной девятой наших обучающих данных. Это может стоить того, если мы хотим попробовать другие улучшения. Кроме того, нужно решить вопрос с тестовым набором, который перетянет на себя часть и без того ограниченных обучающих данных.

Кроме того, стоит внимательно изучить случаи, когда сеть работает не так хорошо, как хотелось бы, и попробовать определить в них закономерность. Но помимо этого, кратко обсудим способы улучшения проекта. В каком-то смысле данный раздел похож на раздел 8.5 главы 8. Мы постараемся подбросить вам идеи, которые вы можете попробовать реализовать. Не страшно, если не все они будут вам досконально понятны¹.

14.7.1. Борьба с переобучением: выбор лучшей регуляризации

На протяжении всей части II в каждой из трех задач (в классификаторах в главе 11 и в разделе 14.5, а также в сегментации в главе 13) в моделях возникало переобучение. В первом случае оно было катастрофическим, и мы решили проблему, сбалансировав и дополнив данные в главе 12. Балансировка данных для предотвращения переобучения побудила нас применить U-Net-архитектуру для изучения небольших областей вокруг узелков и кандидатов, а не полных срезов. В остальных случаях переобучения мы просто прекратили обучение раньше, чем переобучение начало влиять на результаты проверки. Это означает, что предотвращение или снижение переобучения — одно из направлений, в которых можно поискать возможности улучшения.

Сам рабочий шаблон «получить более подходящую модель, а затем снижать переобучение» — неплохой рецепт². Таким образом, этот двухэтапный процесс можно использовать, когда нужно улучшить состояние, которого мы достигли сейчас.

Классическая регуляризация и дополнение

Вы могли заметить, что мы даже не использовали все методы регуляризации из главы 8. Например, можно было бы попробовать выполнить отсев.

Кое-какое дополнение мы выполняли, но можно пойти дальше. Существует относительно эффективный метод увеличения, который мы не пытались

¹ По крайней мере, один из авторов хотел бы написать целую книгу по темам, затронутым в этом разделе.

² См. также сообщение в блоге Андрея Карпати (Andrej Karparthy) A Recipe for Training Neural Networks на <https://karpathy.github.io/2019/04/25/recipe>.

использовать, — эластичная деформация, в которой мы вводим во входные данные «цифровые деформации»¹. Это обеспечивает гораздо бóльшую вариативность, чем просто вращение и поворот, и выглядит применимым к нашей задаче.

Более абстрактное дополнение

До этого момента дополнение данных было вдохновлено геометрией — мы преобразовали наши входные данные так, чтобы они выглядели как нечто правдоподобное. Оказывается, не обязательно ограничиваться этим типом дополнения.

Вспомним из главы 8, что математически потери перекрестной энтропии, которые мы использовали, являются мерой несоответствия между двумя распределениями вероятностей: распределением предсказаний и распределением, которое ставит на метку всю массу вероятности и может быть представлено для метки одномерным вектором. Если сеть становится чрезмерно самоуверенной, то мы можем попробовать просто не применять однократное распределение, а вместо этого присвоить небольшую вероятностную массу «неправильным» классам². Это называется *сглаживанием меток*.

Кроме того, мы можем работать с входными данными и метками одновременно. Для этого была предложена очень общая, а также простая в применении техника аугментации под названием «смешивание»³: авторы предлагают случайным образом интерполировать как входные данные, так и метки. Интересно, что в случае предположения о линейности потерь (которое удовлетворяется бинарной перекрестной энтропией) это эквивалентно простому манипулированию входными данными с весами, взятыми из соответствующим образом адаптированного распределения⁴. Очевидно, что от реальных данных мы не ожидаем смешанности, но такое смешение, по всей видимости, способствует стабильности прогнозов и очень эффективно.

Ансамбли: зачем ограничиваться одной моделью

Существует взгляд на проблему переобучения, который заключается в том, что наша модель могла бы работать так, как мы хотим, если бы мы знали правильные параметры (но мы их не знаем)⁵. Следуя этому интуитивному призыву,

¹ Вы найдете рецепт (хотя и направленный на TensorFlow) по ссылке <http://mng.bz/Md5Q>.

² Для этого можно использовать `nn.KLDivLoss`.

³ Zhang H. et al. mixup: Beyond Empirical Risk Minimization, <https://arxiv.org/abs/1710.09412>.

⁴ См. пост Ференца Хусара (Ferenc Huszar): <http://mng.bz/aRJj/>, в котором также есть код для PyTorch.

⁵ Хотя мы могли бы полноценно воспользоваться теорией байесовской вероятности, мы лишь поверим интуиции.

мы могли бы попытаться поработать с несколькими наборами параметров (то есть несколькими моделями), надеясь, что недостатки одной модели могут компенсировать недостатки другой. Этот метод оценки нескольких моделей и объединения результатов называется *ансамблем*. Проще говоря, мы обучаем несколько моделей, а потом запускаем их все и усредняем предсказания. Когда каждая отдельная модель переобучается (или если мы остановили модель непосредственно перед началом переобучения), они могут начать делать неверные прогнозы на разных входных данных, а не всегда сначала подгонять одни и те же сохраненные данные.

Для создания ансамбля обычно используются совершенно отдельные прогоны обучения или вовсе различные структуры моделей. Но если бы мы хотели упростить себе жизнь, мы могли бы сделать несколько снимков модели из одного тренировочного прогона, причем желательно незадолго до конца или до того, как мы начнем наблюдать признаки переобучения. Затем можно построить ансамбль этих снимков, но поскольку они все равно будут довольно похожи друг на друга, их можно усреднить. Данный подход лежит в основе идеи *стохастического усреднения веса*¹. При этом следует проявлять некоторую осторожность: например, при использовании в модели пакетной нормализации. Мы можем захотеть скорректировать статистику, но, скорее всего, сможем достичь повышения точности даже без этого.

Обобщение знаний, приобретаемых сетью

Можно также рассмотреть *многозадачное обучение*, в котором модель должна будет производить дополнительные данные, помимо оцениваемых². Эти данные должны надежно отражать прогресс улучшения результатов. Мы могли бы одновременно выполнять обучение на узелках и не узелках, а также на доброкачественных и злокачественных новообразованиях. На самом деле данные о злокачественных новообразованиях — тоже метки, которые мы могли бы использовать в качестве дополнительной задачи (об этом чуть позже). Сама идея тесно связана с концепцией трансферного обучения, которую мы рассмотрели ранее, но здесь мы обычно обучаем обе задачи параллельно, а не поочередно.

Если дополнительных задач нет, а запас неразмеченных данных есть, то можно частично реализовать обучение *без учителя*. Недавно был предложен выглядящий довольно эффективным подход: дополнение данных без учителя³. Здесь

¹ Павел Измайлов (Pavel Izmailov) и Эндрю Гордон Уилсон (Andrew Gordon Wilson) представляют введение в код PyTorch: <http://mng.bz/guwe>.

² *Ruder S.* An Overview of Multi-Task Learning in Deep Neural Networks, <https://arxiv.org/abs/1706.05098>, но это также ключевая идея во многих областях.

³ *Xie Q. et al.* Unsupervised Data Augmentation for Consistency Training. <https://arxiv.org/abs/1904.12848>.

мы обучаем нашу модель по данным обычным образом. На размеченных данных делаем прогноз на наборе без дополнения. Затем берем сам прогноз в качестве цели для этого набора и обучаем модель прогнозировать ее также и для дополненной выборки. Другими словами, мы не знаем, верен ли прогноз, но просим сеть выдавать согласованные результаты независимо от того, правильные ли они.

Когда у нас нет ни интересных задач, ни дополнительных данных, мы можем придумать что-нибудь сами. Создавать данные вручную довольно сложно (хотя иногда для этого с некоторым успехом используют генеративные сети, такие как описанные в главе 2), так что вместо этого мы составляем задачи. Это полноценное обучение *без учителя*, а задачи в нем называются *претекстом*. В претекстовых задачах часто встречаются искаженные входные данные, а мы должны обучить сеть реконструировать оригинал (например, с помощью архитектуры, подобной U-Net) или обучить классификатор отличать настоящие данные от искаженных, используя большие части модели (например, сверточные слои).

Впрочем, здесь мы все равно должны придумать способ исказить входные данные. Если мы не хотим этого делать и не получаем желаемых результатов, то обучение без учителя можно реализовать и по-другому. В широком смысле задача будет заключаться в том, чтобы модель нашла достаточно хорошие признаки и научилась различать разные образцы из имеющегося набора данных. Это называется *контрастным обучением*.

Чтобы добавить конкретности, рассмотрим следующий сценарий: мы берем признаки, извлеченные из текущего изображения, и большое количество K других изображений. Это ключевой набор признаков. Теперь поставим задачу классификации следующим образом: учитывая признаки текущего изображения (*запрос*), к какому из $K + 1$ *ключевых* признаков он принадлежит? На первый взгляд задача может показаться тривиальной, но даже при идеальном совпадении между признаками запроса и ключевыми признаками правильного класса обучение будет направлено на то, чтобы признаки запроса максимально отличались от признаков других изображений (для классификатора это выражается в присвоении низкой вероятности). Конечно, тут не хватает множества деталей. Мы рекомендуем вам рассмотреть понятие контраста импульса¹.

14.7.2. Подготовка обучающих данных

Существует несколько способов улучшить обучающие данные. Ранее мы упоминали, что классификация злокачественных новообразований на самом

¹ Xie Q. et al. Momentum Contrast for Unsupervised Visual Representation Learning. <https://arxiv.org/abs/1911.05722>.

деле основана на более тонкой категоризации нескольких рентгенологов. Можно использовать данные, которые мы отбросили, задав по отношению к ним вопрос «Злокачественный или нет?» и выделив пять классов. Затем оценки рентгенологов можно было бы использовать в качестве сглаженной метки: мы могли бы закодировать каждую из них, а затем усреднить для данного узелка. Таким образом, если четыре радиолога смотрят на узелок и двое называют его «неопределенным», третий — «умеренно подозрительным», а четвертый — «крайне подозрительным», то мы выполняем обучение с функцией перекрестной энтропии между выходными данными модели и целевым значением и распределением вероятностей, заданным вектором $0\ 0\ 0.5\ 0.25\ 0.25$. Это было бы похоже на сглаживание меток, которое мы использовали ранее, но конкретизированное для конкретной задачи. Однако нам придется найти новый способ оценки работы модели, поскольку понятия простой точности, ROC и AUC, которые были в бинарной классификации, применять уже нельзя.

Еще один способ использовать несколько оценок — выполнить обучение нескольких моделей вместо одной, причем каждую обучить на аннотациях, предоставленных одним рентгенологом. В конце мы объединим модели, например усреднив их выходные вероятности.

Продолжаем тему разделения на несколько задач. Обратимся снова к данным аннотации, предоставленным PyLIDC, где у каждой аннотации есть дополнительные классификации (тонкость, внутренняя структура, кальцификация, сферичность, четкость границ, дольчатость, спекуляция и текстура (<https://pyliddc.github.io/annotation.html>)). Но для работы с такими критериями вам придется прочесть об узелках больше информации.

В задаче сегментации можно сравнить работу масок, предоставленных PyLIDC, и наших собственных масок. Поскольку данные LIDC аннотированы несколькими рентгенологами, можно было бы сгруппировать узелки в группы «высокой степени согласия» и «низкой степени согласия». Было бы интересно посмотреть, насколько сложно придется классификатору, то есть правильно ли он классифицирует простые случаи, затрудняясь лишь там, где люди тоже сомневались. Можно также подойти к проблеме с другой стороны, определив, насколько трудно обнаруживать узелки с точки зрения производительности модели: «легкие» (правильно классифицированные после одной или двух эпох обучения), «средние» (в конечном итоге классифицированные правильно) и «сложные» (постоянно классифицируемые неправильно).

Помимо использования легкодоступных данных, возможно, стоит выделить категории узелков по типу злокачественности. Если задействовать специалиста, который подробно изучит обучающие данные и укажет тип рака для каждого узелка, а затем научить модель определять данный тип, обучение может стать

более качественным. Стоимость такой работы, разумеется, непомерно высока для хобби-проектов, но в коммерческих условиях это вполне реально.

Особо сложные случаи можно направлять экспертам для проверки на наличие ошибок. Опять же, это требует денег, но в серьезной работе это нормально.

14.7.3. Итоги конкурса и научные работы

Целью части II книги было пройти весь путь от проблемы до решения, и мы это сделали. Но над проблемой обнаружения и классификации узелков в легких люди работали и до нас, поэтому, если хотите обладать более полной информацией, стоит ознакомиться с их достижениями.

Data Science Bowl 2017

В части II мы работали только с КТ-сканами из набора данных LUNA. Однако много информации по этой теме можно найти в Data Science Bowl 2017 (www.kaggle.com/c/data-science-bowl-2017) от Kaggle (www.kaggle.com). Сами данные скачать уже нельзя, но есть статьи от других людей, где они описывают, что у них сработало, а что — нет. Например, некоторые из финалистов Data Science Bowl (DSB) отмечали, что во время обучения полезной оказалась подробная информация об уровне злокачественности (1 ... 5) из LIDC.

Два основных момента, на которые стоит обратить внимание¹:

- решение, занявшее второе место, принадлежит Даниэлю Хаммаку (Daniel Hammack) и Джулиану де Виту (Julian de Wit): <http://mng.bz/Md48>;
- описание решения на девятом месте от команды Deep Breath: <http://mng.bz/aRAX>.

ПРИМЕЧАНИЕ

Многие из новых методов, на которые мы намекали ранее, не были доступны участникам DSB. Пять лет, которые прошли между DSB 2017 и выходом этой книги в печать, — целая вечность в сфере глубокого обучения!

Чтобы проверка выполнялась более правильно, можно было бы использовать для нее набор данных DSB вместо повторного применения нашего набора. К сожалению, DSB перестали делиться необработанными данными, поэтому, если не удастся получить старую копию, вам понадобится другой источник данных.

¹ Спасибо ресурсы Internet Archive за помощь.

Работы LUNA

В ходе LUNA Grand Challenge было получено несколько многообещающих результатов (<https://luna16.grand-challenge.org/Results>). Не во всех статьях приведена довольно подробная информация, чтобы можно было воспроизвести результаты, но во многих из них достаточно информации для дальнейшего развития проекта. Вы можете просмотреть ряд работ и попытаться воспроизвести подходы, которые кажутся интересными.

14.8. ИТОГИ ГЛАВЫ

Этой главой мы завершаем часть II и выполняем обещание, данное в главе 9: теперь у нас есть работающая сквозная система, которая пытается диагностировать рак легких по компьютерной томографии. Если взглянуть на то, с чего мы начали, то мы увидим позади долгий путь, и, надеемся, мы многому научились в процессе.

Мы научили модель делать что-то интересное и сложное с помощью общедоступных данных. Ключевой вопрос таков: «Пригодится ли это для чего-нибудь в реальном мире?» А за ним следует еще один: «Готов ли наш продукт к производству?» Определение *производства* в данном случае зависит от *предполагаемого использования*, так что если вопрос в том, может ли наш алгоритм заменить эксперта-радиолога, то ответ — нет. Это скорее некая самая исходная и сырая версия инструмента, который мог бы в будущем помочь рентгенологу в повседневной клинической практике. Например, подобный инструмент может быть источником второго мнения о чем-то, что могло остаться незамеченным.

Применение подобного инструмента потребует одобрения компетентных регулирующих органов (таких, как Управление по санитарному надзору за качеством пищевых продуктов и медикаментов в Соединенных Штатах), иначе его нельзя использовать для чего-либо, помимо исследований. Чего нам, безусловно, не хватает, так это большого и тщательно отобранного набора данных для дальнейшего обучения и, что еще более важно, проверки нашей работы. Некоторые случаи должны оцениваться несколькими экспертами в контексте всего исследования пациента. В таких случаях нужно учитывать множество ситуаций, от простых до крайних случаев.

Все возможные применения, от чисто исследовательского до клинической проверки и клинического использования, вынуждают работать с моделью в пригодной для масштабирования среде. Излишне говорить, что это связано с собственным набором проблем, как технических, так и с точки зрения процесса. Мы обсудим некоторые технические проблемы в главе 15.

14.8.1. За кулисами

Покончив с моделированием части II, мы хотим немного приоткрыть завесу тайны о том, как на самом деле происходит работа над проектами в сфере глубокого обучения. По сути, в этой книге представлен несколько искаженный взгляд на вещи, поскольку вам был дан готовый тщательно подобранный набор препятствий и возможностей — эдакая ухоженная садовая дорожка через джунгли глубокого обучения. Мы думаем, что серия задач (особенно в части II) сделает книгу лучше, и надеемся, что это поможет вашему обучению в будущем. Но о *реалистичном* опыте здесь речь не идет.

Скорее всего, подавляющее большинство из ваших экспериментов окончится неудачей. Не каждая идея превращается в открытие, и не каждое изменение становится прорывом. Глубокое обучение — это непросто. Оно непостоянно. Помните также, что глубокое обучение — передний фронт наших знаний, новая граница, которую мы исследуем и наносим на карту каждый день *прямо сейчас*. Сейчас удачный момент для того, чтобы попасть в тренд, но, конечно, придется слегка попотеть.

Ради прозрачности приведем пару вещей, которые у нас не удались и не сработали или, по крайней мере, сработали недостаточно хорошо, чтобы их можно было оставить:

- использование функции `HardTanh` вместо `Softmax` для сети классификации (это было проще объяснить, но идея не сработала);
- попытка исправить проблемы, вызванные функцией `HardTanh`, путем усложнения классификационной сети (пропуска соединений и т. д.);
- неудачная инициализация весов приводит к нестабильности обучения, особенно в задаче сегментации;
- обучение сегментации на полных срезах КТ;
- взвешивание потерь для сегментации с помощью SGD. Это не сработало, пришлось брать Adam;
- настоящая 3D-сегментация КТ. У нас не сработала, но DeepMind справился¹. Это было до того, как мы перешли к обрезке фрагментов для узелков, и у нас закончилась память, так что вы можете попробовать еще раз, но уже в текущем варианте;
- неправильное понимание смысла столбца `class` из данных LUNA, из-за которого пришлось переписывать часть книги;

¹ *Nikolov S. et al.* Deep Learning to Achieve Clinically Applicable Segmentation of Head and Neck Anatomy for Radiotherapy. <https://arxiv.org/pdf/1809.04430.pdf>.

- случайно забыли убрать хак «Я хочу получить результаты побыстрее», который отбросил 80 % узелков-кандидатов, найденных модулем сегментации, из-за чего результаты выглядели ужасно, пока мы не поняли, в чем дело (это стоило целых выходных!);
- множество различных оптимизаторов, функций потерь и архитектур моделей;
- балансировка обучающих данных различными способами.

Это еще не все, но остальное мы успели забыть. Многое пошло не так, прежде чем все заработало! Пожалуйста, учитесь на наших ошибках.

Стоит также добавить, что для многих задач в этой книге мы просто выбрали подход. Мы *ни в коем случае* не говорим, что другие подходы хуже (многие из них, вероятно, немного лучше!). Кроме того, стиль кодирования и дизайн проекта у каждого свой.

В машинном обучении очень многие программируют в Jupyter Notebooks. Документы Jupiter — отличный инструмент для быстрого экспериментирования, но у них есть свои недостатки: например, невозможность отслеживать свои действия. Наконец, вместо того, чтобы использовать механизм кэширования `precache`, мы могли бы иметь отдельный этап предварительной обработки, на котором данные записывались бы в виде последовательности тензоров. Каждый из этих подходов — дело вкуса; даже среди трех авторов каждый из нас поступил бы по-разному¹. Всегда полезно попробовать более удачное решение, сохраняя гибкость сотрудничества с вашими коллегами.

14.9. УПРАЖНЕНИЯ

1. Реализуйте тестовый набор для классификации или повторно задействуйте набор из упражнений в главе 13. Используйте проверочный набор для выбора лучших эпох во время обучения, а тестовый — для оценки сквозного проекта. Насколько хорошо производительность проверочного набора соответствует производительности тестового?
2. Сможете ли вы обучить единую модель, которая может выполнять трехстороннюю классификацию и отличать не узелки, доброкачественные узелки и злокачественные узелки за один раз?

¹ О, какие у нас были дискуссии!

- А. Какой вариант балансировки лучше всего подходит для обучения?
- Б. Как работает эта однопроходная модель по сравнению с двухпроходным подходом, который мы используем в книге?
3. Мы обучили наш классификатор на аннотациях, но ожидаем, что он будет работать на входных данных после сегментации. Используйте модель сегментации, чтобы создать список не узелков, и примените их для обучения вместо готовых не узелков.
 - А. Улучшается ли производительность модели классификации при обучении на этом новом наборе?
 - Б. Можете ли вы сказать, какие типы кандидатов в узелки после такого обучения модели изменились больше других?
4. Используемые нами дополненные свертки не полностью восстанавливают контекст вблизи краев изображения. Вычислите потери для сегментированных пикселей вблизи краев среза КТ по сравнению с теми, которые находятся внутри. Есть ли ощутимая разница?
5. Попробуйте запустить классификатор на всей КТ, используя перекрывающиеся фрагменты размером $32 \times 48 \times 48$. Как это соотносится с подходом сегментации?

14.10. РЕЗЮМЕ

- Четкое разделение между обучающими и проверочными (и тестовыми) наборами невероятно важно. Подход с разделением по пациентам гораздо менее подвержен ошибкам. Это особенно верно, когда в конвейере используется несколько моделей.
- Переход от пиксельных меток к узелкам осуществляется с помощью традиционной обработки изображений. Мы не хотим смотреть свысока на классику, но ценим эти инструменты и используем их там, где это уместно.
- Наш сценарий диагностики выполняет как сегментацию, так и классификацию. Это позволяет нам ставить диагноз по КТ, которую мы раньше не видели, хотя текущая реализация `Dataset` не умеет принимать `series_uids` из источников, отличных от LUNA.
- Тонкая настройка — прекрасный способ довести модель до ума, используя минимум обучающих данных. Убедитесь, что у предварительно обученной модели есть признаки, соответствующие вашей задаче, и переобучаемая часть сети обладает достаточной пропускной способностью.

- Инструмент **TensorBoard** позволяет создавать множество различных типов диаграмм, которые помогут нам понять, что происходит. Но это не замена ручному анализу данных, на которых модель работает особенно плохо.
- Успешное обучение, по-видимому, на каком-то этапе обязательно доходит до переобучения, которое затем нужно устранить. Это тоже можно принять как рецепт. К тому же, нам следует больше узнать о регуляризации.
- Обучение нейронных сетей заключается в следующем: пробовать что-то, видеть, что идет не так, и чинить это. Панацеи здесь не бывает.
- **Kaggle** — отличный источник идей для проектов глубокого обучения. Авторы многих наборов данных предлагают денежные призы для лучших исполнителей, а в старых конкурсах есть примеры, которые можно использовать в качестве отправной точки для дальнейших экспериментов.

Часть III

Развертывание

В части III мы рассмотрим, как довести модель до уровня, когда ее можно будет использовать. В предыдущих частях мы научились строить модели: в части I познакомились со сборкой и обучением моделей, а в части II подробно рассмотрели пример от начала до конца; так что тяжелая работа позади.

Но никакая модель не будет полезной, пока вы не сможете применить ее в реальном мире. Поэтому теперь нужно где-то разместить модели и применить их к задачам, для решения которых они предназначены. Эта часть по духу ближе к части I, поскольку в ней представлено множество компонентов PyTorch. Как и прежде, мы сосредоточимся на приложениях и задачах, которые хотим решить, а не просто на PyTorch.

В отдельной главе части III мы познакомимся с ландшафтом развертывания PyTorch по состоянию на начало 2020 года. Мы познакомимся с JIT-компилятором PyTorch и с его помощью экспортируем модели в целях использования в сторонних приложениях к C++ API для поддержки мобильных устройств.

15

Развертывание в производстве

В этой главе

- ✓ Варианты развертывания моделей PyTorch.
- ✓ Работа с PyTorch JIT.
- ✓ Развертывание сервера моделей и экспорт моделей.
- ✓ Запуск экспортированных и встроенных моделей из C++.
- ✓ Запуск моделей на мобильном телефоне.

В части I мы много узнали о моделях, а в части II прошли полный путь создания хороших моделей для решения задачи. Теперь же, когда модели готовы, их нужно применить там, где они будут полезны. Поддержание инфраструктуры для исследования моделей глубокого обучения в масштабе может быть эффективным как с точки зрения архитектуры, так и с точки зрения затрат. Платформа PyTorch изначально ориентирована на исследования, но, начиная с версии 1.0, появились функции, ориентированные на производство, и сегодня PyTorch стала превосходной сквозной платформой для исследований и крупномасштабного производства.

Сам термин «развертывание в производстве» может иметь разные значения в зависимости от задачи.

- Возможно, самым естественным вариантом развертывания моделей, которые мы разработали в части II, была бы настройка сетевого сервиса, через который можно было бы обращаться к моделям. Мы сделаем это в двух вариантах, используя облегченные веб-фреймворки Python: Flask (<http://flask.pocoo.org>) и Sanic (<https://sonicframework.org>). Первый — один из самых популярных в своем классе, а второй идейно похож на него, но задействует новые функции Python `async/await` и асинхронные операции, что позволяет повысить эффективность.
- Мы можем экспортировать модель в хорошо стандартизированный формат, что поможет поставлять ее через оптимизированные процессоры для работы с моделями, специализированное оборудование или облачные сервисы. Для моделей PyTorch можно использовать формат Open Neural Network Exchange (ONNX).
- Может понадобиться интегрировать модель в более крупные приложения. Для этого придется выйти за рамки чистого Python. Чтобы достичь данной цели, мы изучим работу с моделями PyTorch из C++, учитывая, что рассмотренные подходы можно применять к любому языку.
- Наконец, в ряде задач наподобие «зебрафикации» изображений, которую мы рассматривали в главе 2, бывает полезно запустить модель на мобильном устройстве. Конечно, маловероятно, что модуль компьютерной томографии будет работать с мобильным телефоном, но есть и другие медицинские приложения, такие как ручной скрининг кожи, и в подобных приложениях пользователю было бы удобнее работать с мобильным устройством, а не отправлять фрагмент кожи в облачный сервис. К счастью для нас, в PyTorch недавно появилась поддержка мобильных устройств, и мы исследуем это.

Изучив, как реализовать все эти механизмы, мы попробуем взять классификатор из главы 14 и развернуть его на сервере, а затем переключимся на модель «зебрафикации» и на ее примере изучим остальное.

15.1. ПОСТАВКА МОДЕЛЕЙ PYTORCH

Посмотрим, что нам нужно для размещения нашей модели на сервере. Мы продолжим придерживаться практического подхода и начнем с самого простого из возможных серверов. Как только простой и базовый вариант заработает, мы рассмотрим его недостатки и попытаемся их устранить. Наконец, рассмотрим вещи, которые на момент написания статьи лишь планировались. Создадим в сети уютное место для модели¹.

¹ Чтобы не рисковать, не делайте этого в ненадежной сети.

15.1.1. Размещение модели на сервере Flask

Flask — один из наиболее широко используемых модулей Python. Его можно установить простой командой¹:

```
pip install Flask
```

API библиотеки реализуется через декораторы функций (листинг 15.1).

Листинг 15.1. flask_hello_world.py:1

```
from flask import Flask
app = Flask(__name__)

@app.route("/hello")
def hello():
    return "Hello World!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

Запущенное приложение будет работать на порте 8000, и в нем появится один маршрут `/hello`, который возвращает строку `Hello World!`. На данном этапе мы можем расширить наш сервер Flask, загрузив ранее сохраненную модель и передав ее запросом `POST`. В качестве примера будем использовать классификатор узелков из главы 14.

Для получения данных применим функцию `request` из Flask (несколько любопытно импортированной). Объект `request.files` содержит словарь файловых объектов, проиндексированных по именам полей. Мы будем использовать JSON для анализа ввода и вернем строку JSON с помощью функции `jsonify`.

Вместо `/hello` создадим маршрут `/predict`, который по запросу `POST` в виде файлов принимает на вход двоичный блок данных (содержимое серии в пикселях) и связанные метаданные (объект JSON, содержащий словарь с ключом `shape`) и возвращает ответ JSON с прогнозируемым диагнозом. То есть сервер берет один образец узелка (а не пакет) и возвращает вероятность того, что он злокачественный.

Чтобы получить данные, нам сначала нужно декодировать JSON в двоичный файл, который затем с помощью функции `numpy.frombuffer` можно декодировать в одномерный массив. Его мы преобразуем в тензор, используя функцию `torch.from_numpy`.

Сама обработка модели выполняется так же, как в главе 14: мы создадим экземпляр `LunaModel` из главы 14, загрузим веса, которые получили в результате обучения, и переключим модель в режим `eval`. Поскольку обучение уже не

¹ Или `pip3`, если вы используете Python3. Вы также можете запустить команду из виртуальной среды Python.

выполняется, мы скажем PyTorch, что градиенты при запуске модели не потребуются. Для этого добавим блок `with torch.no_grad()` (листинг 15.2).

Листинг 15.2. flask_server.py:1

```
import numpy as np
import sys
import os
import torch
from flask import Flask, request, jsonify
import json

from p2ch13.model_cls import LunaModel

app = Flask(__name__)

model = LunaModel()
model.load_state_dict(torch.load(sys.argv[1],
                                map_location='cpu')['model_state'])
model.eval()

def run_inference(in_tensor):
    with torch.no_grad():
        # LunaModel берет пакет и выводит кортеж
        out_tensor = model(in_tensor.unsqueeze(0))[1].squeeze(0)
        probs = out_tensor.tolist()
        out = {'prob_malignant': probs[1]}
        return out

@app.route("/predict", methods=["POST"])
def predict():
    meta = json.load(request.files['meta'])
    blob = request.files['blob'].read()
    in_tensor = torch.from_numpy(np.frombuffer(
        blob, dtype=np.float32))
    in_tensor = in_tensor.view(*meta['shape'])
    out = run_inference(in_tensor)
    return jsonify(out)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
    print (sys.argv[1])
```

Создание модели, загрузка весов и настройка режима

Автоградиент выключен

По маршруту /predict ожидается передача данных из формы в POST-запросе

В запросе будет один файл с именем meta

Преобразование данных из двоичного формата в torch

Кодирование ответа в JSON

Запустим сервер:

```
python3 -m p3ch15.flask_server
➡ data/part2/models/cls_2019-10-19_15.48.24_final_cls.best.state
```

Мы подготовили простой клиент в сценарии `cls_client.py`, который отправляет один пример. Из каталога кода вы можете запустить его с помощью команды:

```
python3 p3ch15/cls_client.py
```

В ответ приложение скажет, что узелок вряд ли будет злокачественным. Итак, наш сервер принимает входные данные, пропускает их через нашу модель и возвращает выходные данные. Получается, все готово? Не совсем. Посмотрим, что можно улучшить.

15.1.2. Требования к развертыванию

Соберем все необходимое для поставки моделей¹. Прежде всего, мы хотим поддерживать *современные протоколы и их особенности*. HTTP старой школы глубоко последователен, то есть, когда клиент хочет отправить несколько запросов в одном и том же подключении, следующий запрос отправляется только после того, как будет получен ответ на предыдущий. Звучит не очень эффективно, если нужно передать пакет чего-либо. Мы частично решим проблему, перейдя на фреймворк Sanic, в котором вопросам эффективности уделяется немало внимания.

При использовании графических процессоров часто гораздо эффективнее обрабатывать запросы *пакетами*, чем запускать их один за другим или параллельно. Итак, нам необходимо взять запросы из нескольких подключений, собрать их в пакет для запуска на ГП, а затем вернуть результаты соответствующим запрашивающим сторонам. Звучит сложно, и кажется, что в простых учебниках так не делают (по крайней мере сейчас). И именно поэтому мы сделаем это! Однако обратите внимание: пока длительность запуска самой модели не стала проблемой (в том смысле, что задержка запуска системы — это нормально, но ожидание выполнения обработки пакета, зависящее от ожидания запроса, — неприемлемо), у нас нет особых причин запускать несколько пакетов на одном графическом процессоре. Эффективнее было бы просто увеличить размер пакета.

Мы хотим обрабатывать несколько пакетов *параллельно*. Даже при асинхронном выполнении нам нужно, чтобы наша модель эффективно работала во втором потоке, то есть нам требуется обойти печально известную глобальную блокировку интерпретатора Python (global interpreter lock, GIL) во время работы модели.

Кроме того, мы хотим *как можно меньше* копировать данные. Многократное копирование данных — это плохо как с точки зрения потребления памяти, так и с точки зрения затрат времени. Многие вещи в HTTP кодируются в Base64 (формат, ограниченный шестью битами на байт и предназначенный для

¹ Одним из первых публичных выступлений, где использование Flask для моделей PyTorch ставилось под сомнение, было: PyTorch under the Hood Кристиана Пероне (Christian Perone). <http://mng.bz/xWdW>.

кодирования двоичного кода в буквенно-цифровые строки), поэтому декодирование изображений в двоичный файл, затем снова в тензор, а затем в пакет будет затратно по времени. Мы частично решим эту проблему с помощью потоковой передачи запросов PUT, чтобы не выделять новые строки Base64, а дополнять их (что ужасно для производительности строк и тензоров). Мы говорим «частично», поскольку полностью избавиться от копирования это не поможет.

Последнее, что желательно для поставки модели, — это *безопасность*. В идеальном мире у нас было бы реализовано безопасное декодирование. Мы хотим защититься как от переполнения, так и от исчерпания ресурсов. Если размер входного тензора известен, то все должно работать хорошо, поскольку сложно сломать PyTorch, работая с входными данными фиксированного размера. Но, чтобы добиться этого через декодирование изображений и т. п., скорее всего, вам придется постараться, и мы не даем никаких гарантий. Интернет-безопасность — достаточно обширная область, поэтому мы вообще не будем ее затрагивать. Отметим, что нейронные сети известны своей восприимчивостью к изменению входных данных и могут генерировать в ответ на них неправильные или непредвиденные выходные данные (известные как *враждебные примеры*), но для нашего приложения это не столь важно, так что опустим данную тему.

Довольно слов. Начнем совершенствовать сервер.

15.1.3. Пакетная обработка запросов

Во втором варианте сервера воспользуемся фреймворком Sanic (который устанавливается через одноименный пакет Python). Это даст нам возможность выполнять множество запросов параллельно с помощью асинхронной обработки, что снимает с нас одну проблему. Попутно реализуем пакетную обработку запросов.

Асинхронное программирование всегда пугает новичков, и в нем обычно очень много терминологии. Но в данном случае мы просто делаем так, чтобы функции не блокировали выполнение в процессе ожидания результатов вычислений или событий¹.

Чтобы реализовать пакетную обработку запросов, мы должны отделить саму обработку запросов от запуска модели. На рис. 15.1 показана схема потока данных.

¹ Пижоны называют эти асинхронные функции генераторами или сопрограммами: <https://en.wikipedia.org/wiki/Coroutine>.

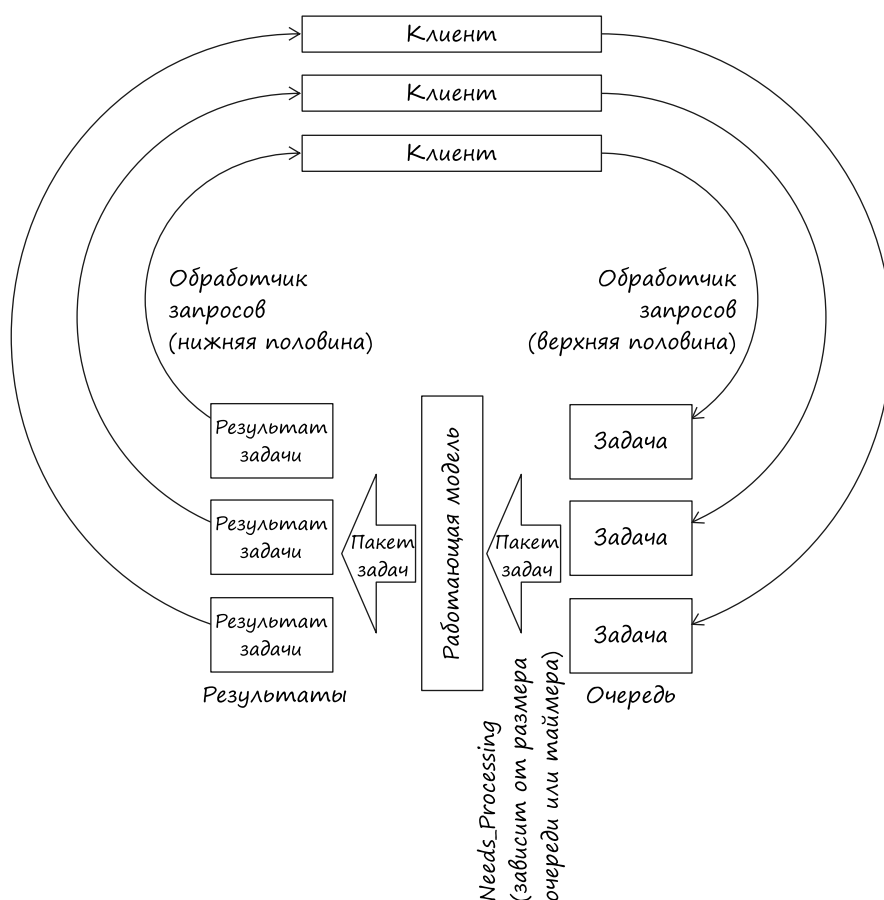


Рис. 15.1. Перемещение данных при пакетной обработке запросов

В верхней части рис. 15.1 показаны клиенты, делающие запросы, которые один за другим проходят через верхнюю половину обработчика запросов. Задачи с информацией о запросе попадают в очередь. Когда в очереди формируется полный пакет или прошло заданное максимальное время ожидания для самого старого запроса, модель берет пакет из очереди, обрабатывает его и выдает результат. Затем они по одному обрабатываются нижней половиной процессора запросов.

Реализация

Реализация будет содержать две функции. Функция, отвечающая за прогон модели, запускается сразу и работает вечно. Всякий раз, когда нам нужно запустить модель, она собирает пакет входных данных, запускает модель во втором потоке (чтобы программа могла делать что-то другое) и возвращает результат.

Затем процессор запросов декодирует запрос, ставит входные данные в очередь, ожидает завершения обработки и возвращает выходные данные с результатами. Чтобы понять смысл слова «*асинхронный*», представьте корзину для бумаги. Все рисунки, которые мы набросали для этой главы, выбрасываются в корзину справа от стола. Ее нужно опустошать, но лишь время от времени: либо когда она заполняется, либо когда пришло время вечерней уборки. Точно так же мы ставим в очередь новые запросы, при необходимости запускаем обработку и ждем результатов, прежде чем отправлять их в качестве ответа на запрос. На рис. 15.2 обе функции показаны в блоках, которые мы непрерывно выполняем перед передачей управления обратно в цикл обработки событий.

Небольшая сложность этой схемы заключается в том, что у нас есть два случая, когда нам нужно обработать события: если мы накопили полную партию или прошло достаточно времени ожидания для самого старого запроса. Для второго случая нужен таймер¹.

Весь интересный код помещен в класс `ModelRunner`, как показано в листинге 15.3.

Листинг 15.3. `request_batching_server.py:32, ModelRunner`

```
class ModelRunner:
    def __init__(self, model_name):
        self.model_name = model_name
        self.queue = []  # ← Очередь

        self.queue_lock = None  # ← Это блокировка

        self.model = get_pretrained_model(self.model_name,
                                          map_location=device)  # ←

        self.needs_processing = None  # ← Сигнал для запуска модели

        self.needs_processing_timer = None  # ← Наконец, таймер
```

Загрузка и создание экземпляра модели. Это единственное, что нам нужно изменить для перехода на J1T. На данный момент мы импортируем CycleGAN (с небольшой модификацией для подгонки ввода и вывода под диапазон [0... 1]) из файла `p3ch15/cyclegan.py`

Класс `ModelRunner` сперва загружает модель и выполняет некоторые административные задачи. Помимо модели, нам также нужно несколько других вещей. Мы добавляем наши запросы в очередь `queue`. Это просто список Python, в котором мы добавляем задачи в конце и удаляем в начале.

Изменяя `queue`, мы хотим сделать так, чтобы другие задачи не могли менять ее в тот же момент. Для этого введем `queue_lock` — экземпляр `asyncio.Lock` из модуля `asyncio`. Объекты `asyncio`, которые мы здесь используем, должны знать цикл событий, доступный только после инициализации приложения, поэтому в момент создания экземпляра временно зададим его как `None`. Подобная

¹ Как вариант, можно отказаться от таймера и запускать модель, только когда очередь не пуста. Это потенциально может привести к запуску мелких «первых» пакетов, но общее влияние на производительность в большинстве приложений будет невелико.

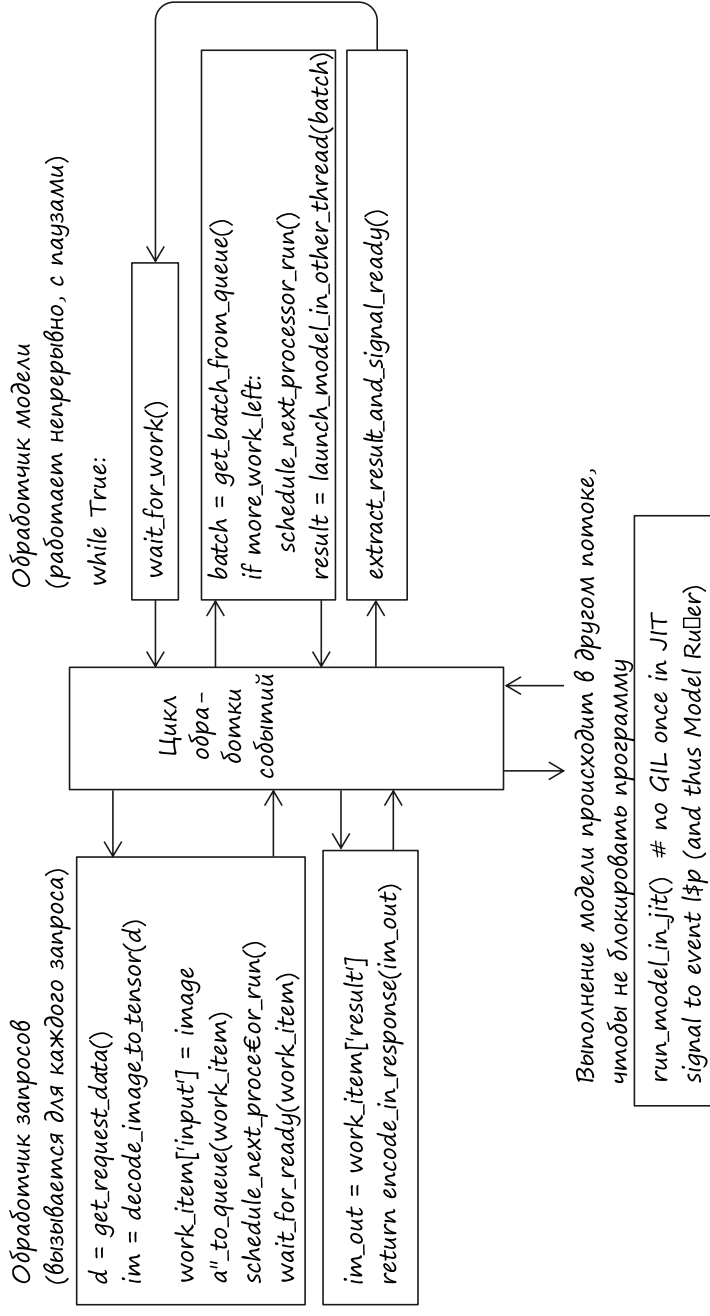


Рис. 15.2. Асинхронный сервер состоит из трех блоков: обработка запросов, обработка модели и выполнения модели. Эти блоки немного похожи на функции, но первые два передают управление циклу обработки событий

блокировка может и не быть строго обязательной, поскольку наши методы не возвращаются в цикл обработки событий и держат блокировку, а операции с очередью атомарны благодаря GIL, но зато мы в коде изложили наше видение процесса. Если бы у нас было несколько исполнителей, то блокировка точно потребовалась бы. Одно предостережение: асинхронные блокировки Python не являются потокобезопасными (увы).

Когда классу `ModelRunner` нечего делать, он просто ждет. Нам нужно подать ему сигнал от `RequestProcessor`, чтобы он проснулся и приступил к работе. Это делается через `asyncio.Event` под названием `need_processing`. `ModelRunner` использует метод `wait()` и ждет события `need_processing`. `RequestProcessor` использует метод `set()`, отправляет сигнал, а `ModelRunner` просыпается и очищает событие методом `clear()`.

Наконец, нам нужен таймер, ограничивающий максимальное время ожидания. Этот таймер создается методом `app.loop.call_at`. Он активирует событие `need_processing` — пока мы зарезервируем место под него. Иногда событие будет активироваться напрямую по заполнении пакета, а иногда по таймеру. Когда мы обрабатываем пакет до того, как сработает таймер, мы очищаем его, чтобы не выполнять лишнюю работу.

От запроса к очереди

Теперь нам нужна возможность ставить запросы в очередь, то есть первая часть `RequestProcessor` на рис. 15.2 (без декодирования и перекодирования). Это делается в асинхронном методе `process_input` (листинг 15.4).

Листинг 15.4. `request_batching_server.py:54`

```

async def process_input(self, input):
    our_task = {"done_event": asyncio.Event(loop=app.loop),  ← Данные для задачи
               "input": input,
               "time": app.loop.time()}
    async with self.queue_lock:  ← Активируем блокировку,
                                добавляем задачу и...
        if len(self.queue) >= MAX_QUEUE_SIZE:
            raise HandlingError("I'm too busy", code=503)
        self.queue.append(our_task)
        self.schedule_processing_if_needed()
    await our_task["done_event"].wait()  ← Ожидание (и возврат в цикл
    return our_task["output"]             с помощью await) завершения обработки

```

...планируем ее обработку. Если соберется полный пакет, то возникнет событие `need_processing`. Если мы этого не сделаем и таймер не установлен, то событие произойдет по истечении максимального времени ожидания

Мы создали небольшой словарь Python, в котором хранится информация о задаче: `input`, `time` в очереди и `done_event`, которое активируется после обработки задачи. Позже добавляется `output`.

Удерживая блокировку очереди (это удобно делать в блоке `async with`), мы добавляем нашу задачу в очередь и при необходимости планируем обработку.

В качестве меры предосторожности мы поднимаем ошибку, если очередь становится слишком большой. Тогда нам остается лишь дождаться обработки задачи и вернуть ее.

ПРИМЕЧАНИЕ

Важно использовать время цикла (как правило, монотонные часы), которое может отличаться от `time.time()`. В противном случае события могут оказаться запланированными до попадания в очередь или вообще не обработаться.

Это все, что нам нужно для обработки запроса (кроме декодирования и кодирования).

Запуск пакета из очереди

Далее посмотрим на функцию `model_runner` в правой части рис. 15.2, которая запускает модель (листинг 15.5).

Листинг 15.5. `request_batching_server.py:71, .run_model`

```
async def model_runner(self):
    self.queue_lock = asyncio.Lock(loop=app.loop)
    self.needs_processing = asyncio.Event(loop=app.loop)
    while True:
        await self.needs_processing.wait()  # ← Ожидание задач
        self.needs_processing.clear()
        if self.needs_processing_timer is not None:  # ← Отмена таймера, если он установлен
            self.needs_processing_timer.cancel()
            self.needs_processing_timer = None
        async with self.queue_lock:
            # ... строка 87
            to_process = self.queue[:MAX_BATCH_SIZE]  # ← Захват пакета и планирование
            del self.queue[:len(to_process)]           # ← запуска следующего пакета,
            self.schedule_processing_if_needed()         # если это необходимо
        batch = torch.stack([t["input"] for t in to_process], dim=0)
        # здесь мы можем удалить ввод...

        result = await app.loop.run_in_executor(
            None, functools.partial(self.run_model, batch)  # ← Запуск модели в отдельном
        )                                                    # потоке, перемещение данных
        for t, r in zip(to_process, result):                # и передача их модели.
            t["output"] = r                                  # Мы продолжаем обработку
            t["done_event"].set()                             # после того, как это будет сделано
        del to_process

        # ← Добавление результатов и запуск
        # соответствующего события
```

Как показано на рис. 15.2, `model_runner` выполняет настройку, а затем бесконечно закидывается (передавая управление циклу обработки событий). Она вызывается при создании экземпляра приложения, поэтому может настроить `queue_lock` и событие `need_processing`, о котором мы говорили ранее. Затем запускается цикл в ожидании события `need_processing`.

Когда приходит событие, мы сначала проверяем, установлено ли время, и если да, то очищаем его, поскольку запускается обработка. Затем `model_runner` берет пакет из очереди и при необходимости планирует обработку следующего пакета. Он собирает пакет из отдельных задач и запускает новый поток работы модели с помощью функции `app.loop.run_in_executor` из `asyncio`. Наконец, мы добавляем выходные данные к задачам и активируем `done_event`.

В принципе, все. Веб-фреймворку, который похож на Flask, но умеет работать с `async` и `await`, нужна небольшая обертка. Нужно запускать функцию `model_runner` в цикле событий. Как упоминалось ранее, блокировка очереди не требуется, если у нас нет нескольких исполнителей, которые берут задачи из очереди и потенциально могут прерывать друг друга. Но, зная, что наш код будет адаптирован для других проектов, мы заложим эту возможность.

Запустим сервер с помощью команды:

```
python3 -m p3ch15.request_batching_server data/p1ch2/horse2zebra_0.4.0.pth
```

Теперь мы можем проверить его работу на изображении `data/p1ch2/horse.jpg` и сохранить результат:

```
curl -T data/p1ch2/horse.jpg  
➡ http://localhost:8000/image --output /tmp/res.jpg
```

Обратите внимание, что у данного сервера появились хорошие возможности (он группирует запросы для графического процессора и работает асинхронно), но мы по-прежнему используем Python, поэтому GIL мешает запуску модели параллельно с обработкой запросов в основном потоке. Такой механизм небезопасен для потенциально враждебных сред наподобие интернета. В частности, декодирование данных запроса не кажется ни оптимальным по скорости, ни полностью безопасным.

Было бы лучше, если бы декодирование выполнялось в момент передачи запроса в функцию вместе с предварительно выделенным фрагментом памяти и функция декодировала изображение из потока. Но мы не знаем библиотеки, которая помогла бы нам в этом.

15.2. ЭКСПОРТ МОДЕЛИ

Все это время мы использовали PyTorch из интерпретатора Python. Но это не всегда желательно: GIL по-прежнему будет блокировать работу веб-сервера. Нам может понадобиться запускать программу на встроенных системах, где Python слишком дорог или вообще отсутствует. Пришло время экспортировать модель. Существует несколько способов сделать это. Мы можем полностью отказаться от PyTorch и перейти на более специализированные фреймворки. А можем

остаться в экосистеме PyTorch и использовать JIT (just-in-time), компилятор для ориентированного на PyTorch подмножества Python. JIT-модель в Python обладает двумя преимуществами: JIT иногда позволяет выполнять отличные оптимизации, а также, как в случае с нашим веб-сервером, дает нам возможность просто обойти GIL. Наконец (но нам потребуется некоторое время, чтобы до этого дойти), мы можем запустить нашу модель с помощью `libtorch`, библиотеки C++ от PyTorch, с производной от нее `Torch Mobile`.

15.2.1. Совместимость за пределами PyTorch с ONNX

Иногда хочется сохранить вокруг модели экосистему PyTorch, например, для запуска на встроенном оборудовании со специализированным конвейером развертывания модели. Для этой цели Open Neural Network Exchange предоставляет интерпретационный формат для нейронных сетей и моделей машинного обучения (<https://onnx.ai>). После экспорта модель может выполняться в любой совместимой с ONNX среде выполнения, например ONNX Runtime¹, при условии, что операции, используемые в нашей модели, поддерживаются стандартом ONNX и целевой средой выполнения. Так, на Raspberry Pi это работает немного быстрее, чем напрямую с PyTorch. Помимо традиционного оборудования, множество специализированных аппаратных ускорителей искусственного интеллекта поддерживает ONNX (<https://onnx.ai/supported-tools.html#deployModel>).

В каком-то смысле модель глубокого обучения — программа с очень специфическим набором инструкций, состоящая из отдельных операций, таких как умножение матриц, свертка, `relu`, `tanh` и т. п. Следовательно, если мы можем сериализовать вычисление, то можем повторно выполнить его в другой среде выполнения, которая понимает входящие в него низкоуровневые операции. ONNX — стандартизация формата, описывающего эти операции и их параметры.

Большинство современных фреймворков глубокого обучения поддерживают сериализацию вычислений в ONNX, и часть из них могут загружать файл ONNX и выполнять его (хотя это не относится к PyTorch). Некоторые малогабаритные («граничные») устройства принимают файлы ONNX в качестве входных данных и генерируют низкоуровневые инструкции для конкретного устройства. Некоторые поставщики облачных вычислений позволяют загружать файл ONNX и просматривать его через конечную точку REST.

¹ Код находится в репозитории <https://github.com/microsoft/onnxruntime>, но перед работой с ним обязательно прочитайте заявление о конфиденциальности! В настоящее время сборка ONNX Runtime производит пакет, который не отправляет данные в материнское хранилище.

Чтобы экспортировать модель в ONNX, нам нужно запустить модель с фиктивными входными данными: значения входных тензоров не столь важны, главное, чтобы они были правильной формы и типа. С помощью функции `torch.onnx.export` PyTorch *отслеживает* вычисления, выполняемые моделью, и сериализует их в файл ONNX с указанным именем:

```
torch.onnx.export(seg_model, dummy_input, "seg_model.onnx")
```

Полученный файл ONNX можно запустить в среде выполнения, скомпилировать на граничное устройство или загрузить в облачный сервис. Его можно использовать из Python после установки `onnxruntime` или `onnxruntime-gpu` и получить пакет как массив NumPy (листинг 15.6).

Листинг 15.6. `onnx_example.py`

```
import onnxruntime

sess = onnxruntime.InferenceSession("seg_model.onnx")
input_name = sess.get_inputs()[0].name
pred_onnx, = sess.run(None, {input_name: batch})
```

API среды выполнения ONNX использует сеансы для определения моделей и вызывает метод запуска с именovanными входами. Это довольно типичная схема при работе с вычислениями, определенными в статических графах

Не все операторы TorchScript можно представить как стандартные операторы ONNX. Если мы экспортируем операции, которые ONNX чужды, то получаем ошибки о неизвестных операторах `aten`, когда пытаемся использовать среду выполнения.

15.2.2. Встроенный механизм экспорта PyTorch: отслеживание

Когда совместимость не столь важна, но все еще нужно обойти Python GIL или иным образом экспортировать сеть, мы можем использовать собственное представление PyTorch, называемое *Torch-Script graph*. Ниже мы увидим, что это такое и как работает JIT, который его генерирует. Давайте попробуем.

Самый простой способ создать модель TorchScript — отследить ее. Этот механизм работает точно так же, как экспорт ONNX. Оно и неудивительно, поскольку «под капотом» тоже используется модель ONNX. Здесь мы просто подаем в модель фиктивные входные данные с помощью функции `torch.jit.trace`. Мы импортируем `UNetWrapper` из главы 13, загружаем обученные параметры и переводим модель в режим оценки.

Прежде чем выполнить отслеживание модели, дадим еще одно предостережение: ни один из параметров не должен требовать градиентов, поскольку применение

контекстного менеджера `torch.no_grad()` переключает их использование во время выполнения. Даже если мы проследим модель внутри `no_grad`, а затем запустим ее снаружи, PyTorch запишет градиенты. На рис. 15.4 видно, почему это так: после отслеживания модели мы просим PyTorch выполнить ее. Но у отслеживаемой модели при выполнении записанных операций будут параметры, требующие градиентов. Чтобы избежать этого, нам пришлось бы запускать трассируемую модель в контекст `torch.no_grad`. Чтобы избавиться от этого (о таких вещах легко забыть, а затем долго удивляться отсутствию производительности), мы перебираем параметры модели и делаем так, чтобы градиенты нигде не требовались.

Тогда нам достаточно вызвать `torch.jit.trace`¹ (листинг 15.7).

Листинг 15.7. `trace_example.py`

```
import torch
from p2ch13.model_seg import UNetWrapper

seg_dict = torch.load('data-unversioned/part2/models/p2ch13/seg_2019-10-20_15
➔.57.21_none.best.state', map_location='cpu')
seg_model = UNetWrapper(in_channels=8, n_classes=1, depth=4, wf=3,
➔ padding=True, batch_norm=True, up_mode='upconv')
seg_model.load_state_dict(seg_dict['model_state'])
seg_model.eval()
for p in seg_model.parameters():  ← Задаем такие параметры, чтобы градиенты не требовались
    p.requires_grad_(False)

dummy_input = torch.randn(1, 8, 512, 512)
traced_seg_model = torch.jit.trace(seg_model, dummy_input)  ← Отслеживание
```

Получаем предупреждение:

```
TracerWarning: Converting a tensor to a Python index might cause the trace
to be incorrect. We can't record the data flow of Python values, so this
value will be treated as a constant in the future. This means the trace
might not generalize to other inputs!
    return layer[:, :, diff_y:(diff_y + target_size[0]), diff_x:(diff_x +
➔ target_size[1])]
```

Предупреждение связано с обрезкой, которую мы делали в U-Net, но пока мы планируем загружать в модель только изображения размером 512×512 , все будет в порядке. В следующем подразделе мы более подробно рассмотрим, откуда берется предупреждение и как обойти ограничение, на которое оно указывает. Это также будет важно, если будет нужно преобразовывать в TorchScript более сложные модели, чем сверточные и U-сети.

¹ Строго говоря, модель отслеживается как функция. Недавно в PyTorch появилась возможность сохранять большую часть структуры модуля, используя `torch.jit.trace_module`, но нам хватит и простого отслеживания.

Мы можем сохранить отслеженную модель:

```
torch.jit.save(traced_seg_model, 'traced_seg_model.pt')
```

и загрузить ее обратно с сохраненным файлом:

```
loaded_model = torch.jit.load('traced_seg_model.pt')
prediction = loaded_model(batch)
```

PyTorch JIT запомнит состояние модели в момент сохранения, то есть настройку в режим оценки и отключение требования градиентов. Если бы мы не позаботились об этом заранее, то нам нужно было бы использовать контекст `with torch.no_grad():`.

СОВЕТ

Вы можете запустить экспортированную JIT-модель PyTorch, не сохраняя исходный код. Но нам всегда хочется иметь такой рабочий процесс, при котором мы автоматически переходим от исходной модели к установленной JIT-модели для развертывания. Если мы этого не сделаем, то при необходимости что-то исправить в модели мы утратим способность изменять и регенерировать ее. Всегда сохраняй исходники, юный падаван!

15.2.3. Сервер с отслеженной моделью

Сейчас нужно обновить веб-сервер до окончательной версии. Мы можем экспортировать отслеженную модель CycleGAN следующим образом:

```
python3 p3ch15/cyclegan.py data/p1ch2/horse2zebra_0.4.0.pth
⇒ data/p3ch15/traced_zebra_model.pt
```

Теперь нам просто нужно заменить вызов функции на `get_pretrained_model` с `torch.jit.load` (и убрать ненужный импорт `get_pretrained_model`). Это также означает, что модель работает независимо от GIL, а именно этого мы и хотели добиться от сервера. Для вашего удобства мы внесли небольшие изменения в файл `request_batching_jit_server.py`. Мы можем запустить его, передав путь к файлу отслеживания в качестве аргумента командной строки.

Теперь, когда мы прочувствовали всю силу JIT, углубимся в детали!

15.3. ВЗАИМОДЕЙСТВИЕ С PYTORCH JIT

Появившийся в PyTorch 1.0 инструмент PyTorch JIT стал центром нововведений в PyTorch, не последним из которых является богатый инструментарий для развертывания.

15.3.1. Что за пределами Python/PyTorch

Довольно часто говорят, что Python не хватает скорости. В этом есть доля правды, но операции с тензорами, которые мы используем в PyTorch, обычно сами по себе достаточно сложные, поэтому медлительность Python на их фоне не выглядит большой проблемой. Для небольших устройств, таких как смартфоны, накладные расходы Python выглядят более значительно. Поэтому имейте в виду, что часто ускорение, получаемое за счет исключения Python из вычислений, составляет 10 % или меньше.

Еще одно преимущество отказа от Python с точки зрения скорости проявляется в многопоточных средах, и тогда ускорение получается значительным. Поскольку промежуточные значения не являются объектами Python, на вычисления не влияет угроза параллелизации Python и GIL. Это то, что мы уже видели ранее, когда использовали на сервере отслеживаемую модель.

Уход от классического последовательного выполнения операций в PyTorch позволяет PyTorch получать целостное представление о процессе вычисления и знать о нем все. Это дает возможность проводить важные оптимизации и преобразования более высокого уровня. Одни из них применяются в основном для вывода, а другие также могут обеспечить значительное ускорение обучения.

Рассмотрим короткий пример, который даст вам представление о том, почему просмотр нескольких операций одновременно может оказаться полезным. Когда PyTorch выполняет последовательность операций на графическом процессоре, он вызывает подпрограмму (*ядро*, на языке CUDA) для каждой из них. Каждое ядро считывает входные данные из памяти графического процессора, вычисляет результат, а затем сохраняет его. Таким образом, большая часть времени обычно тратится не на вычисления, а на чтение и запись в память. Это можно улучшить, выполнив чтение один раз, вычислив несколько операций и затем записав результат в самом конце. Это именно то, что делает PyTorch JIT. Чтобы вам стало понятнее, как это работает, на рис. 15.3 показаны точечные вычисления, происходящие в долговременной памяти (LSTM; https://en.wikipedia.org/wiki/Long_short-term_memory), которая часто используется в рекуррентных сетях.

Детали рис. 15.3 нам здесь неважны, но интерес представляют пять входных данных вверху, два выхода внизу и семь промежуточных результатов, представленных в виде округленных индексов. Вычисляя все это за один раз в одной функции CUDA и сохраняя промежуточные значения в регистрах, JIT сокращает количество операций чтения памяти с двенадцати до пяти и количество операций записи с девяти до двух. Это большой выигрыш, который позволяет сократить время обучения сети LSTM в четыре раза.

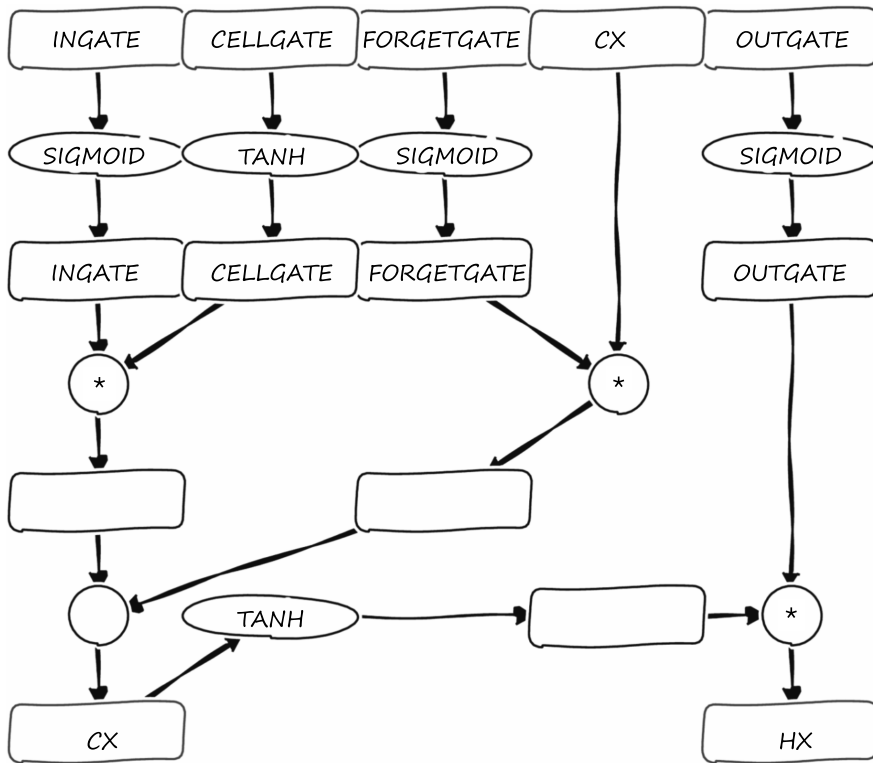


Рис. 15.3. Точечные операции с ячейками LSTM. Данный блок из пяти входных элементов вычисляет два выходных. Промежуточные поля — это промежуточные результаты, которые оригинальный PyTorch сохраняет в памяти, а JIT держит в регистрах

Этот, казалось бы, простой трюк позволяет PyTorch значительно сократить разрыв между скоростью LSTM и обобщенных ячеек LSTM, гибко определенных в PyTorch, и жесткой, но высокооптимизированной реализацией LSTM из библиотек вроде cuDNN.

Подводя итог, можно сказать, что ускорение от использования JIT и исключения их вычислений Python не столь велико, чем можно было бы ожидать, когда говорят, что Python ужасно медленный. А вот отказ от GIL — это серьезный шаг в многопоточных приложениях. Значительное ускорение в JIT-моделях достигается за счет специальных оптимизаций, предусмотренных в JIT, но реализация в данном случае явно сложнее, чем простой обход накладных расходов Python.

15.3.2. Двойственная природа PyTorch как интерфейса и бекэнда

Чтобы понять, как работает PyTorch за пределами Python, можно мысленно разделить его на несколько частей. Мы впервые увидели это в разделе 1.4. Модуль PyTorch `torch.nn`, который мы впервые увидели в главе 6 и который был нашим основным инструментом для моделирования, хранит параметры сети и реализуется с помощью функционального интерфейса: функций, принимающих и возвращающих тензоры. Они реализованы в виде расширения C++, переданного слою с поддержкой автоградации уровня C++ (затем фактические вычисления передаются внутренней библиотеке ATen, выполняющей вычисления, но это неважно).

Учитывая, что функции C++ уже готовы, разработчики PyTorch просто превратили их в официальный API. Получилось ядро LibTorch, позволяющее нам писать на C++ операции с тензорами, которые выглядят почти так же, как их аналоги на Python. Модуль `torch.nn` по природе своей предназначен только для Python, C++ API отражает их в пространстве имен `torch::nn`, которое очень похоже на Python, но независимо от него.

Это позволило бы нам воспроизвести на C++ то, что мы уже делали на Python. Но хотим-то мы не этого. Мы хотим *экспортировать* модель. К счастью, есть еще один интерфейс для тех же функций PyTorch: PyTorch JIT. Он дает «символическое» представление вычислений — *промежуточное представление TorchScript* (TorchScript IR, а иногда просто TorchScript). Мы упоминали TorchScript в подразделе 15.2.2, когда говорили об отложенных вычислениях. В следующих разделах мы увидим, как получить это представление наших моделей Python и как их сохранять, загружать и выполнять. Подобно тому, что мы обсуждали для обычного API PyTorch, JIT-функции PyTorch для загрузки, проверки и выполнения модулей TorchScript также могут быть доступны как из Python, так и из C++.

Таким образом, на рис. 15.4 показаны четыре способа вызова функций PyTorch: как из C++, так и из Python мы можем либо вызывать функции напрямую, либо использовать JIT в качестве посредника. Все эти способы в конечном итоге вызывают функции C++ LibTorch, а оттуда — ATen и вычислительный бэкэнд.

15.3.3. TorchScript

TorchScript находится в центре инструментов развертывания, предусмотренных PyTorch. Поэтому нам стоит внимательно исследовать, как он работает.

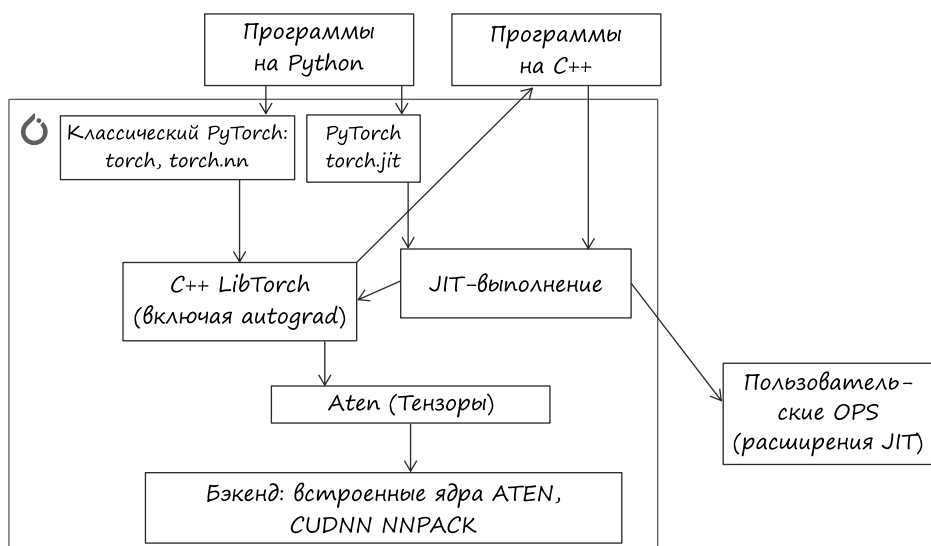


Рис. 15.4. Множество способов обращения к PyTorch

Существует два простых способа создания модели TorchScript: отслеживание и создание сценариев. Мы рассмотрим каждый из них в следующих подразделах. На очень высоком уровне они работают следующим образом.

- *Отслеживание*, которое мы использовали в подразделе 15.2.2, — это выполнение обычной модели PyTorch на случайных входных данных. В PyTorch JIT имеет хуки (в интерфейсе автоградации C++) для каждой функции, которые записывают выполняемые вычисления. В каком-то смысле это все равно что сказать: «Посмотрите, как я вычисляю результаты, — теперь вы можете сделать то же самое». Учитывая, что JIT вступает в игру только тогда, когда работает PyTorch (а также `nn.Modules`), вы можете запускать любой код Python во время трассировки, но JIT заметит только эти биты (и, в частности, не будет знать о потоке управления). Когда мы используем тензорные формы — обычно кортеж целых чисел, — JIT пытается следить за тем, что происходит, но, возможно, не сможет понять до конца. Поэтому мы получили предупреждение при отслеживании U-Net.
- *Сценарии* PyTorch JIT просматривают фактический код Python, отвечающий за вычисления, и компилируют его в TorchScript IR. Это означает, что JIT точно охватит все тонкости программы, но лишь в той мере, в какой код будет понятен компилятору. Это все равно что сказать: «Я говорю вам, как это сделать, а теперь вы делаете то же самое». Звучит как программирование, если подумать.

Но мы сюда пришли заниматься не теорией, так что попробуем оба варианта на примере очень простой функции, которая неэффективно выполняет сложение по первому измерению:

```
# In[2]:
def myfn(x):
    y = x[0]
    for i in range(1, x.size(0)):
        y = y + x[i]
    return y
```

Мы можем отследить ее:

```
# In[3]:
inp = torch.randn(5,5)
traced_fn = torch.jit.trace(myfn, inp)
print(traced_fn.code)
```

```
# Out[3]:
def myfn(x: Tensor) -> Tensor:
    y = torch.select(x, 0, 0)  ← Индексация первой строки функции
    y0 = torch.add(y, torch.select(x, 0, 1), alpha=1)  ← Цикл развернут и работает
    y1 = torch.add(y0, torch.select(x, 0, 2), alpha=1) с числами от 1 до 4 независимо
    y2 = torch.add(y1, torch.select(x, 0, 3), alpha=1) от размера x
    _0 = torch.add(y2, torch.select(x, 0, 4), alpha=1)
    return _0
```

TracerWarning: Converting a tensor to a Python index might cause the trace to be incorrect. We can't record the data flow of Python values, so this value will be treated as a constant in the future. This means the trace might not generalize to other inputs! ←
 Страшно, но что поделать

Мы видим большое предупреждение, связанное с тем, что в коде используется фиксированная индексация и сложение пяти строк, то есть код не будет работать должным образом с четырьмя или шестью строками.

Сценарии решают проблему:

```
# In[4]:
scripted_fn = torch.jit.script(myfn)
print(scripted_fn.code)

# Out[4]:
def myfn(x: Tensor) -> Tensor:
    y = torch.select(x, 0, 0)
    _0 = torch.__range_length(1, torch.size(x, 0), 1)  ← PyTorch строит диапазон
    y0 = y  из размера тензора
    for _1 in range(_0):  ← Цикл for, в котором используется странный код
        i = torch.__derive_index(_1, 1, 1) для получения индекса i
        y0 = torch.add(y0, torch.select(x, 0, i), alpha=1)  ← Тело цикла, которое немного
    return y0 вносит ясность
```


Мы также можем вывести граф сценария, который будет ближе к внутреннему представлению TorchScript:

```
# In[5]:
xprint(scripted_fn.graph)
# end::cell_5_code[]

# tag::cell_5_output[]
# Out[5]:
graph(%x.1 : Tensor):
  %10 : bool = prim::Constant[value=1]() ← Кажется, подробностей многовато
  %2 : int = prim::Constant[value=0]()
  %5 : int = prim::Constant[value=1]()
  %y.1 : Tensor = aten::select(%x.1, %2, %2) ← Первое присвоение y
  %7 : int = aten::size(%x.1, %2)
  %9 : int = aten::__range_length(%5, %7, %5) ← Создание диапазона
  → %y : Tensor = prim::Loop(%9, %10, %y.1) ← стало выглядеть яснее
    block0(%i1 : int, %y.6 : Tensor):
      %i.1 : int = aten::__derive_index(%i1, %5, %5)
      %18 : Tensor = aten::select(%x.1, %2, %i.1) ← Тело цикла for: выбор среза
      %y.3 : Tensor = aten::add(%y.6, %18, %5) ← и его прибавление к y
      → (%10, %y.3)
    return (%y)

Цикл for возвращает значение y,
которое вычисляет
```

На практике `torch.jit.script` чаще всего используется в виде декоратора:

```
@torch.jit.script
def myfn(x):
    ...
```

Вы также можете сделать это с помощью пользовательского декоратора `trace`, который бы обрабатывал входные данные, но этот метод не прижился.

Хотя TorchScript (язык) выглядит как подмножество Python, между ними есть фундаментальные различия. Если мы посмотрим очень внимательно, то увидим, что в PyTorch в коде есть спецификации типов. Это намекает на важное различие: TorchScript статически типизирован, то есть каждое значение (переменная) в программе имеет один и только один тип. Кроме того, используются только те типы, которые понимает TorchScript IR. Внутри программы JIT обычно определяет тип автоматически, но нам необходимо аннотировать любые нетензорные аргументы функций их типами. Все это резко контрастирует с Python, где мы можем присвоить что угодно чему угодно.

Пока что механизм сценариев мы опробовали только на функциях. Но мы уже давно, еще в главе 5, перешли от простого использования функций к модулям. Разумеется, мы можем вдобавок отслеживать или создавать сценарии для модулей. В результате они будут вести себя примерно так же, как обычные

и горячо любимые нами модули. И для трассировки, и для сценариев мы передаем экземпляр `Module` в `torch.jit.trace` (с любыми входными данными) или в `torch.jit.script` (без входных данных) соответственно. В результате получится метод `forward`, к которому мы привыкли. Если мы хотим раскрыть другие методы (это работает только при создании сценариев), то к ним в определении класса можно добавить декоратор `@torch.jit.export`.

Когда мы сказали, что JIT-модули работают так же, как и в Python, мы имели в виду еще и то, что их можно использовать и для обучения. С другой стороны, это означает, что нам нужно настроить их для логического вывода (например, с помощью контекста `torch.no_grad()`) так же, как традиционные модели, чтобы все работало правильно.

Если модель относительно проста алгоритмически, как CycleGAN или модели классификации и сегментация на основе U-Net, то мы можем просто отследить ее. У более сложных моделей есть особенность: мы можем использовать скриптовые или отслеживаемые функции из другого скриптового или отслеживаемого кода, а также применять скриптовые или отслеживаемые подмодули при построении и отслеживании или скриптинге модуля. Мы также можем отслеживать функции, вызывая `nn.Models`, но тогда нам нужно настроить все параметры так, чтобы они не требовали градиентов, поскольку параметры отслеживаемой модели будут постоянными.

Поскольку отслеживание мы уже обсудили, более пристально рассмотрим практический пример написания сценариев.

15.3.4. Использование сценариев как лучшей замены отслеживания

В более сложных моделях, используемых при обработке естественного языка, таких как модели семейства Fast R-CNN, или в рекуррентных сетях биты, отвечающие за поток управления, например циклы, должны записываться в сценарий. Аналогично, если бы нам нужна была гибкость, мы бы нашли бит кода, о котором предупреждал отслеживатель (листинг 15.8).

Листинг 15.8. Из `utils/unet.py`

```
class UNetUpBlock(nn.Module):
    ...
    def center_crop(self, layer, target_size):
        _, _, layer_height, layer_width = layer.size()
        diff_y = (layer_height - target_size[0]) // 2
        diff_x = (layer_width - target_size[1]) // 2
        return layer[:, :, diff_y:(diff_y + target_size[0]),
        ➡ diff_x:(diff_x + target_size[1])]  ← Здесь будет выдано предупреждение

    def forward(self, x, bridge):
```

```
...
    crop1 = self.center_crop(bridge, up.shape[2:])
...
```

Здесь JIT волшебным образом заменяет кортеж формы `up.shape` одномерным целочисленным тензором с той же информацией. Теперь отслеживается только срез `[2:]` и вычисление `diff_x` и `diff_y`. Однако это нас не спасает, поскольку для срезов нужен тип `int` из Python, а здесь полномочия JIT уже заканчиваются.

Но мы можем решить эту проблему простым способом: создадим сценарий для `center_crop`. Мы меняем место взятия среза, передавая `up` в `center_crop` и выполняя срез уже там. Помимо этого, осталось лишь добавить декоратор `@torch.jit.script`. Результатом является следующий код, который позволяет отслеживать модель U-Net без предупреждений (листинг 15.9).

Листинг 15.9. Переписанный отрывок из `utils/unet.py`

```
@torch.jit.script
def center_crop(layer, target):  ← Изменение подписи — берем target вместо target_size
    _, _, layer_height, layer_width = layer.size()
    _, _, target_height, target_width = target.size()  ← Получаем размеры
                                                         внутри сценария
    diff_y = (layer_height - target_height) // 2
    diff_x = (layer_width - target_width) // 2
    return layer[:, :, diff_y:(diff_y + target_height),
    → diff_x:(diff_x + target_width)]  ← Индексация по полученным
                                         значениям размера

class UNetUpBlock(nn.Module):
    ...

    def forward(self, x, bridge):
        ...
        crop1 = center_crop(bridge, up)  ← Соответствующие изменения
                                         в вызове функции
    ...
```

Другой вариант, к которому мы могли бы прибегнуть (но не будем), — переместить то, что нельзя обернуть в сценарий, в пользовательские операторы, реализованные в C++. В библиотеке TorchVision это делается для некоторых специальных операций в моделях Mask R-CNN.

15.4. LIBTORCH: PYTORCH В C++

Мы рассмотрели разные способы экспорта моделей, но во всех по-прежнему использовали Python. Теперь мы рассмотрим, как можно отказаться от Python и работать напрямую с C++.

Вернемся к примеру с превращением лошади в зебру с помощью CycleGAN. Теперь мы возьмем JIT-модель из подраздела 15.2.3 и запустим ее из программы на C++.

15.4.1. Запуск JIT-моделей из C++

Самая сложная часть развертывания визуальных моделей PyTorch на C++ — это выбор библиотеки для работы с изображениями¹. Здесь мы используем очень легкую библиотеку CImg (<http://cimg.eu>). Если вы хорошо знакомы с библиотекой OpenCV, то можете адаптировать код под работу с ней. Мы решили, что CImg для этой задачи подходит больше всего.

JIT-модель запускается очень просто. Сначала мы покажем обработку изображения. На самом деле это не то, что нам нужно, поэтому мы сделаем все очень быстро² (листинг 15.10).

Листинг 15.10. `cyclegan_jit.cpp`

```
#include "torch/script.h"
#define cimg_use_jpeg
#include "CImg.h"
using namespace cimg_library;
int main(int argc, char **argv) {
    CImg<float> image(argv[2]);
    image = image.resize(227, 227);
    // ... здесь нам нужно создать выходной тензор из входного
    CImg<float> out_img(output.data_ptr<float>(), output.size(2),
        output.size(3), 1, output.size(1));
    out_img.save(argv[3]);
    return 0;
}
```

Включает в себя заголовок сценария PyTorch и CImg с поддержкой JPEG

Загрузка и декодирование изображения в массив чисел с плавающей запятой

Уменьшение размера изображения

Сохранение изображения

Метод `data_ptr<float>()` возвращает указатель на хранилище тензоров. С помощью этой информации и информации о форме мы можем построить выходное изображение

Со стороны PyTorch нужно включить C++-заголовок `torch/script.h`. Затем нам нужно настроить и включить библиотеку CImg. В функции `main` мы загружаем изображение из файла, указанного в командной строке, и изменяем его размер (в CImg). Итак, теперь в переменной `CImg<float> image` хранится изображение размером 227×227 пикселей. В конце программы мы создадим `out_img` такого же типа из тензора (1, 3, 277, 277) и сохраним его.

О мелочах думать не надо. Это не тот PyTorch C++, который мы изучаем, так что можно просто взять данный код как есть.

Вычисления тоже довольно просты. Нам нужно сделать входной тензор из изображения, загрузить модель и пропустить входной тензор через нее (листинг 15.11).

¹ Но в TorchVision может появиться и своя функция для загрузки изображений.

² Код работает в PyTorch 1.4 и, надеемся, более новых версиях. В версиях PyTorch до 1.3 вместо `data_ptr` используется `data`.

Листинг 15.11. `cyclegan_jit.cpp`

```

auto input_ = torch::tensor(
    torch::ArrayRef<float>(image.data(), image.size()));
auto input = input_.reshape({1, 3, image.height(),
    image.width()}).div_(255);

auto module = torch::jit::load(argv[1]);

std::vector<torch::jit::IValue> inputs;
inputs.push_back(input);
auto output_ = module.forward(inputs).toTensor();

auto output = output_.contiguous().mul_(255);

```

Помещение данных изображения в тензор

Изменение формы и масштаба для перехода от соглашений CImg к PyTorch

Загрузка JIT-модели или функции из файла

Упаковка ввода в (одноэлементный) вектор IValue

Вызов модуля и извлечение тензора результата. Ради эффективности мы меняем владельца, так как если бы мы держались за IValue, то впоследствии результат был бы пустым

Гарантируем, что результат непрерывный

Вспомним из главы 3, что PyTorch хранит значения тензора в большом фрагменте памяти в определенном порядке. То же самое делает и CImg, и мы можем получить указатель на этот фрагмент памяти (как массив `float`) с помощью метода `image.data()` и количество элементов с помощью метода `image.size()`. В результате получаем более умную ссылку: `torch::ArrayRef` (что является просто сокращением для указателя с размером. PyTorch использует их на уровне данных C++, а также для возврата размеров без копирования). Затем мы можем просто проанализировать конструктор `torch::tensor`, как и в случае со списком.

СОВЕТ

Иногда вместо `torch::tensor` лучше использовать аналогичный конструктор `torch::from_blob`. Разница в том, что `tensor` будет копировать данные. Если вы не хотите копировать, то можете применять `from_blob`, но тогда вам нужно позаботиться о том, чтобы базовая память была доступна в течение всего срока службы тензора.

Тензор у нас одномерный, поэтому нам нужно изменить его форму. Удобно, что в CImg используется тот же порядок, что и в PyTorch (канал, строки, столбцы). Если нет, то нам нужно будет адаптировать изменение формы и переставить оси, как мы делали это в главе 4. Поскольку в CImg используется диапазон `0 ... 255`, а модель работает в диапазоне `0 ... 1`, мы выполним деление и умножение. Конечно, это можно было включить в модель, но мы хотели повторно применить ее.

Загрузить отслеживаемую модель очень просто с помощью конструктора `torch::jit::load`. Далее нам нужно поработать с абстракцией, которую PyTorch вводит для связи между Python и C++: нам нужно обернуть входные данные в `IValue` (или несколько `IValue`), *универсальный* тип данных для любого значения. Функции в JIT передается вектор `IValue`, так что мы указываем это в объявлении и отправляем входной тензор обратно функцией `push_back`. Он окажется обернут в `IValue`. Мы отправляем вектор `IValue` дальше и получаем его назад. Затем можем распаковать тензор в полученное `IValue` с помощью метода `.toTensor`.

РАСПРОСТРАНЕННАЯ ОШИБКА: ПРЕД- И ПОСТОБРАБОТКА

При переходе с одной библиотеки на другую легко забыть проверить совместимость выполняемых преобразований. Они неочевидны, если предварительно не изучить структуру памяти и соглашение о масштабировании PyTorch и библиотеки обработки изображений, которую мы используем. Если мы забудем это сделать, то не получим ожидаемых результатов и расстроимся.

В данном случае модель сойдет с ума, поскольку получит на вход слишком большие входные данные. А соглашение нашей модели в вопросе выходных данных заключается в том, чтобы выдавать значения RGB в диапазоне 0...1. Если бы мы использовали их напрямую с CImg, то результат выглядел бы полностью черным.

У всех фреймворков свои соглашения: например, OpenCV любит хранить изображения в формате BGR вместо RGB, поэтому приходится инвертировать порядок каналов. Всегда нужно проверять, совпадают ли входные данные, которые мы передаем модели в развертывании, с теми, которые мы передаем в нее в Python.

Здесь можно немного узнать об IValue: у них есть тип (в данном случае Tensor), но в них можно хранить и int_64, и double, и список тензоров. Например, если бы у нас было несколько выходов, то мы бы получили IValue со списком тензоров, что следует из соглашений о вызовах Python. Когда мы распаковываем тензор из IValue методом .toTensor, IValue меняет собственность (становится недействительным). Но не стоит беспокоиться об этом, так как мы получили тензор. Иногда модель может возвращать несмежные данные (из-за пробелов в памяти — вспомните главу 3), а CImg обоснованно хочет получить непрерывный блок, мы вызываем функцию contiguous. Мы должны присвоить непрерывный тензор переменной, которая находится в области видимости, пока мы не закончим работу с базовой памятью. Как и в Python, PyTorch освободит память, если увидит, что тензор больше не используется.

Итак, скомпилируем код! В Debian или Ubuntu для работы с CImg вам необходимо установить `cig-dev`, `libjpeg-dev` и `libx11-dev`.

Вы можете загрузить библиотеку C++ PyTorch со страницы PyTorch. Но, учитывая, что у нас уже установлен PyTorch¹, можно взять все нужное прямо из него. Нам нужно знать, где находится наша установка PyTorch, поэтому откройте Python и проверьте значение `torch.__file__`, которое укажет путь наподобие `/usr/local/lib/python3.7/dist-packages/torch/__init__.py`. Это означает, что нужные нам файлы CMake находятся в каталоге `/usr/local/lib/python3.7/dist-packages/torch/share/cmake/`.

¹ Надеемся, что вы не ленитесь и запускаете все примеры.

Применение CMake кажется излишним для проекта из одного файла, но подключить PyTorch немного сложнее. Поэтому мы просто используем приведенный ниже шаблонный код CMake¹ (листинг 15.12).

Листинг 15.12. CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(cyclegan-jit)  ← Имя проекта. Замените его своим именем здесь и далее

find_package(Torch REQUIRED)  ← Нам нужен тип Torch
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${TORCH_CXX_FLAGS}")

add_executable(cyclegan-jit cyclegan_jit.cpp)
target_link_libraries(cyclegan-jit pthread jpeg X11)  ←
target_link_libraries(cyclegan-jit "${TORCH_LIBRARIES}")
set_property(TARGET cyclegan-jit PROPERTY CXX_STANDARD 14)
```

Мы хотим скомпилировать исполняемый файл с именем cyclegan-jit из файла cyclegan_jit.cpp

Ссылки на биты, необходимые для CImg. Сам CImg включает все нужное, поэтому тут не отображается

Лучше всего создать каталог сборки в каталоге с исходным кодом, а затем запустить в нем CMake с помощью команды² `CMAKE_PREFIX_PATH=/usr/local/lib/python3.7/dist-packages/torch/share/cmake/cmake ..` и `make`. Создастся программа `cyclegan-jit`, которую мы затем можем запустить следующим образом:

```
./cyclegan-jit ../traced_zebra_model.pt ../../data/p1ch2/horse.jpg /tmp/z.jpg
```

Мы только что запустили нашу модель PyTorch без Python. Потрясающе! Если вы хотите доставить приложение на сервер, то нужно будет скопировать библиотеки из `/usr/local/lib/python3.7/dist-packages/torch/lib` туда, где находится ваш исполняемый файл, чтобы их всегда можно было найти.

15.4.2. Сразу работаем с C++ и API C++

Модульный API C++ во многом похож на API Python. Чтобы вы это прочувствовали, мы переведем генератор CycleGAN в модель, изначально определенную на C++, но без JIT. Но для этого нужны будут веса из обученной модели, поэтому мы сохраним отслеживаемую версию модели (и здесь важно отслеживать не функцию, а модель).

¹ У каталога с кодом есть более длинная версия для решения проблем с Windows.

² Возможно, вам придется заменить путь тем, где находится ваша установка PyTorch или LibTorch. Обратите внимание, что библиотека C++ может быть более требовательна к совместимости, чем библиотека Python: если вы используете библиотеку с поддержкой CUDA, то вам необходимо установить соответствующие заголовки CUDA. Если вы получаете загадочные сообщения об ошибках «Caffe2 с использованием CUDA», то вам необходимо установить версию библиотеки только для ЦП, но CMake обнаружил версию с поддержкой CUDA.

Начнем с простого: подключения и пространства имен (листинг 15.13).

Листинг 15.13. `cyclegan_cpp_api.cpp`

```
#include <torch/torch.h>
#define cimg_use_jpeg
#include <Cimg.h>
using torch::Tensor;
```

Импорт универсального заголовка `torch/torch.h` и `Cimg`

Писать `torch::Tensor` долго, поэтому мы импортируем имя в основное пространство имен

Глядя на исходный код в файле, мы обнаруживаем, что `ConvTransposed2d` определяется на месте, тогда как в идеале его следует брать из стандартной библиотеки. Проблема здесь в том, что модульный API C++ все еще находится на стадии разработки; а с PyTorch 1.4 готовые модули `ConvTranspose2d` нельзя использовать в `Sequential`, поскольку он принимает необязательный второй аргумент¹. Обычно можно просто пропустить `Sequential`, как мы сделали для Python, но мы хотим, чтобы наша модель имела ту же структуру, что и генератор Python CycleGAN из главы 2.

Рассмотрим остальной код (листинг 15.14).

Листинг 15.14. Остальной код в `cyclegan_cpp_api.cpp`

```
struct ResNetBlock : torch::nn::Module {
    torch::nn::Sequential conv_block;
    ResNetBlock(int64_t dim)
        : conv_block(
            torch::nn::ReflectionPad2d(1),
            torch::nn::Conv2d(torch::nn::Conv2dOptions(dim, dim, 3)),
            torch::nn::InstanceNorm2d(
                torch::nn::InstanceNorm2dOptions(dim)),
            torch::nn::ReLU(/*inplace=*/true),
            torch::nn::ReflectionPad2d(1),
            torch::nn::Conv2d(torch::nn::Conv2dOptions(dim, dim, 3)),
            torch::nn::InstanceNorm2d(
                torch::nn::InstanceNorm2dOptions(dim))) {
        register_module("conv_block", conv_block);
    }

    Tensor forward(const Tensor &inp) {
        return inp + conv_block->forward(inp);
    }
};
```

Инициализация `Sequential`, включая подмодули

Не забывайте регистрировать модули, иначе возможна катастрофа!

Как и следовало ожидать, функция `forward` довольно проста

Как и в Python, мы регистрируем подкласс `torch::nn::Module`. В коде представлен последовательный подмодуль `conv_block`.

¹ Это крупное улучшение по сравнению с PyTorch 1.3, где нам нужно было реализовать собственные модули для `ReLU`, `InstanceNorm2d` и пр.

И так же, как мы это делали в Python, нам нужно инициализировать наши подмодули, в частности `Sequential`. Мы делаем это с помощью оператора инициализации C++. Это похоже на то, как мы создаем подмодули в Python в конструкторе `__init__`.

В отличие от Python, в C++ нет возможностей самоанализа и перехвата, которые позволили бы перенаправить `__setattr__`, чтобы присвоение выполнялось в нужном месте.

Поскольку отсутствие именованных аргументов словами делает спецификацию параметров неудобной, модули (например, тензорные фабричные функции) обычно принимают аргумент `options`. Необязательные ключевые аргументы в Python соответствуют методам объектов параметров, которые мы можем передать. Например, модуль Python `nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=1)`, который нам нужно преобразовать, превращается в `torch::nn::Conv2d(torch::nn::Conv2dOptions(in_channels, out_channels, kernel_size).stride(2).padding(1))`. Это несколько более утомительно, но вы читаете это, так как любите C++ и вас не пугают препятствия, которые приходится преодолевать.

Мы всегда должны следить за тем, чтобы регистрация и присвоение членов были синхронизированы, иначе все будет работать не так, как ожидалось. Например, загрузка и обновление параметров во время обучения будут происходить с зарегистрированным модулем, но фактически вызываемый модуль является членом. Данная синхронизация в Python выполняется за кулисами класса `nn.Module`, в C++ это автоматически не происходит. Если не следить за этим, то можно создать себе проблемы.

В отличие от того, что мы сделали (и должны были сделать!) в Python, нам нужно вызывать на модулях функцию `m->forward(...)`. Некоторые модули также можно вызывать напрямую, но с `Sequential` так нельзя.

Последнее замечание по поводу соглашений о вызовах: в зависимости от того, изменяете ли вы тензоры, предоставляемые функциям¹, тензорные аргументы всегда должны передаваться как `const Tensor&` для тензоров, которые остаются неизменными, или `Tensor`, если произошло изменение. Тензоры должны возвращаться как `Tensor`. Неправильные типы аргументов, такие как неконстантные ссылки (`Tensor&`), могут привести к непонятным ошибкам компилятора.

В основном классе генератора мы будем более точно следовать типичному паттерну C++ API, назвав наш класс `ResNetGeneratorImpl` и передав его в модуль `ResNetGenerator` с помощью макроса `TORCH_MODULE`. Суть в том, что мы в основном хотим обрабатывать модули как ссылки или общие указатели. Обернутый класс это и делает (листинг 15.15).

¹ Это немного размыто, поскольку вы можете создать новый тензор с общей памятью со входом и изменить его на месте, но по возможности лучше так не делать.

Листинг 15.15. ResNetGenerator в cylegan_cpp_api.cpp

```

struct ResNetGeneratorImpl : torch::nn::Module {
    torch::nn::Sequential model;
    ResNetGeneratorImpl(int64_t input_nc = 3, int64_t output_nc = 3,
                        int64_t ngf = 64, int64_t n_blocks = 9) {
        TORCH_CHECK(n_blocks >= 0);
        model->push_back(torch::nn::ReflectionPad2d(3));
        ...
        model->push_back(torch::nn::Conv2d(
            torch::nn::Conv2dOptions(ngf * mult, ngf * mult * 2, 3)
                .stride(2)
                .padding(1)));
        ...
        register_module("model", model);
    }
    Tensor forward(const Tensor &inp) { return model->forward(inp); }
};

TORCH_MODULE(ResNetGenerator);

```

Добавление модулей в контейнер Sequential в конструктор. Это позволяет нам добавить переменную числа модулей в цикл for

Здесь скучный код

Пример работы Options

Создание обертки ResNetGenerator вокруг класса ResNetGeneratorImpl. Как бы архаично это ни казалось, здесь важно, чтобы имена совпадали

Вот и все — мы определили точный аналог модели ResNetGenerator из Python на C++. Теперь нам нужна только функция main, которая будет выполнять загрузку параметров и запускать модель (листинг 15.16). Загрузка изображения с помощью CImg и преобразование изображения в тензор и обратно в изображение выполняются так же, как и в предыдущем разделе. Интересы ради мы будем отображать изображение, а не сохранять его на диск.

Листинг 15.16. cylegan_cpp_api.cpp main

```

ResNetGenerator model;
...
torch::load(model, argv[1]);
...
cimg_library::CImg<float> image(argv[2]);
image.resize(400, 400);
auto input_ =
    torch::tensor(torch::ArrayRef<float>(image.data(), image.size()));
auto input = input_.reshape({1, 3, image.height(), image.width()});
torch::NoGradGuard no_grad;

model->eval();
auto output = model->forward(input);
...
cimg_library::CImg<float> out_img(output.data_ptr<float>(),
    output.size(3), output.size(2),
    1, output.size(1));
cimg_library::CImgDisplay disp(out_img, "See a C++ API zebra!");
while (!disp.is_closed()) {
    disp.wait();
}

```

Создание модели

Загрузка параметров

Объявление защитной переменной эквивалентно использованию контекста torch.no_grad(). Вы можете поместить ее в блок {...}, если нужно ограничить длительность выключения градиентов

Как и в Python, включаем режим eval (для нашей модели это было бы не совсем актуально)

Опять же, мы вызываем функцию forward, а не модель

Отображение изображения. Нам нужно дождаться нажатия клавиши, а не завершать программу немедленно

Интересные изменения заключаются в способе, с помощью которого мы создаем и запускаем модель. Как и ожидалось, мы создаем экземпляр модели, объявляя переменную типа модели. Мы загружаем модель, используя конструктор `torch::load` (здесь важно, что мы обернули ее). Хотя это выглядит очень знакомым для практиков PyTorch, обратите внимание, что это будет работать с сохраненными в JIT файлами, а не с сериализованными словарями состояний Python.

При запуске модели нам нужен аналог конструкции `with torch.no_grad():`. Это достигается созданием экземпляра переменной типа `NoGradGuard` и хранения ее в доступной области памяти, пока градиенты не нужны. Как и в Python, мы устанавливаем модель в режим оценки, вызывая `model->eval()`. На сей раз мы вызываем `model->forward` с нашим входным тензором и в результате получаем тензор — JIT не задействуется, поэтому упаковка и распаковка `IValue` не требуется.

Уф. Писать код C++ для фанатов Python, которыми мы являемся, сродни попытке. Хорошо, что мы пообещали только перенос имеющегося кода, но вообще в LibTorch есть также оптимизаторы, загрузчики данных и многое другое. API прежде всего используются тогда, когда вы хотите создавать модели, и ни JIT, ни Python не подходят.

Для вашего удобства в файле `CMakeLists.txt` приведены инструкции по сборке `cyclegan-cpp-api`, поэтому сборка выполняется точно так же, как в предыдущем разделе.

Мы можем запустить программу с помощью команды:

```
./cyclegan_cpp_api ../traced_zebra_model.pt ../../data/p1ch2/horse.jpg
```

Хотя мы и без того знаем, что получится, не так ли?

15.5. ДОБАВИМ МОБИЛЬНОСТИ

В качестве последнего варианта развертывания модели рассмотрим работу с мобильными устройствами. Если модель нужно перенести на мобильное устройство, то речь обычно идет об Android и/или iOS. В этой книге поработаем с Android.

Работающую на C++ часть PyTorch (LibTorch) можно скомпилировать для Android, после чего можно обратиться к ней из приложения, написанного на Java, с помощью Android Java Native Interface (JNI). Но на самом деле нам нужно всего несколько функций из PyTorch — загрузка JIT-модели, ввод данных в тензоры и `IValue`, прогон их через модель и получение результатов. Чтобы избавить нас от необходимости использовать JNI, разработчики PyTorch поместили эти функции в небольшую библиотеку под названием PyTorch Mobile.

Стандартный способ разработки приложений для Android — использование интегрированной среды разработки Android Studio, поэтому мы поступим так же. Однако существует несколько десятков системных файлов, которые меняются в разных версиях Android. Таким образом, мы сосредоточимся на элементах, превращающих один из шаблонов Android Studio (Java-приложение с Empty Activity) в приложение, которое делает снимок, пропускает его через `zebra-cycleGAN` и отображает результат. Придерживаясь темы книги, в примере мы будем эффективно использовать инструменты Android (а это может оказаться не так приятно, как писать код PyTorch).

Чтобы вдохнуть жизнь в шаблон, нам нужно сделать три вещи. Для начала нужно определить пользовательский интерфейс. Чтобы приложение было максимально простым, сделаем всего два элемента: `TextView` под названием `headline`, который можно нажать, чтобы сделать и преобразовать фото, и `ImageView` для отображения изображения, которое мы называем `image_view`. Задачу создания снимков возложим на приложение камеры (чего вы, скорее всего, не будете делать, чтобы не портить впечатление пользователю) — так наш взгляд не будет замыливаться, поскольку мы занимаемся развертыванием моделей PyTorch.

PyTorch нужно подключить в качестве зависимости. Это делается путем редактирования файла `build.gradle` нашего приложения и добавления в него `pytorch_android` и `pytorch_android_torchvision` (листинг 15.17).

Листинг 15.17. Дополнения в файл `build.gradle`

```
dependencies {
    ...
    implementation 'org.pytorch:pytorch_android:1.4.0'
    implementation 'org.pytorch:pytorch_android_torchvision:1.4.0'
}
```

Раздел `dependencies` должен быть тут. Если нет, то добавьте его внизу

Библиотека `pytorch_android` содержит основные нужные нам вещи

Вспомогательная библиотека `pytorch_android_torchvision` — возможно, несколько нескромно названная по сравнению с ее более крупной сестрой `TorchVision` — содержит несколько утилит для преобразования растровых объектов в тензоры (на момент написания больше ничего не делает)

Нужно добавить отслеживаемую модель в качестве ассета.

Наконец, пора добраться до сути нашего блестящего приложения: класса Java, полученного из `activity`, содержащей наш основной код. Обсудим этот фрагмент здесь. Он начинается с импорта и настройки модели (листинг 15.18).

Выполним импорт из пространства имен `org.pytorch`. В типичном стиле, характерном для Java, мы импортируем `IValue`, `Module` и `Tensor`, что неудивительно, а также класс `org.pytorch.torchvision.TensorImageUtils`, который содержит вспомогательные функции для преобразования между тензорами и изображениями.

Листинг 15.18. MainActivity.java, часть 1

```

...
import org.pytorch.IValue;   ← Как вам импорты?
import org.pytorch.Module;
import org.pytorch.Tensor;
import org.pytorch.torchvision.TensorImageUtils;
...
public class MainActivity extends AppCompatActivity {
    private org.pytorch.Module model;   ← Здесь JIT-модель

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        try {   ← В Java нужно перехватывать исключения
            model = Module.load(assetFilePath(this, "traced_zebra_model.pt")); ←
        } catch (IOException e) {
            Log.e("Zebraify", "Error reading assets", e);   ← Загрузка модели из файла
            finish();
        }
        ...
    }
    ...
}

```

Сначала, конечно же, нужно объявить переменную, содержащую нашу модель. Затем во время запуска приложения (событие `onCreate` в `activity`) мы загрузим модуль с помощью метода `Module.load` из места, указанного в качестве аргумента. Однако есть небольшая сложность: данные приложений предоставляются поставщиком как *ассеты*, к которым сложно обратиться из файловой системы. По этой причине нужен служебный метод `assetFilePath` (взято из примеров PyTorch для Android), копирующий актив в файловую систему. Наконец, в Java нам нужно перехватывать исключения, которые генерирует код, кроме случаев, когда мы хотим (и можем) написать метод, выдающий их.

Получив изображение из приложения камеры с помощью инструмента `Intent` Android, нам нужно пропустить его через модель и отобразить. Это происходит в обработчике события `onActivityResult` (листинг 15.19).

Растровое изображение, которое мы получаем от Android, преобразуется в тензор с помощью функции `TensorImageUtils.bitmapToFloat32Tensor` (статический метод), которая принимает два массива чисел с плавающей запятой `means` и `stds`, а также `bitmap`. Здесь мы указываем среднее значение и стандартное отклонение наших входных данных (набора), которые затем будут сопоставлены с нулевым средним значением и единичным стандартным отклонением, как в преобразовании `TorchVision.Normalize`. Android по умолчанию хранит изображения в диапазоне 0–1, которые модели и нужны, поэтому мы указываем среднее значение 0 и стандартное отклонение 1, чтобы нормализация ничего на самом деле не делала.

Листинг 15.19. MainActivity.java, часть 2

```

Выполняется нормализация, хотя по умолчанию
изображения и так закодированы в диапазоне 0...1,
поэтому преобразование выполнять не нужно:
сдвиг равен 0, а делитель — 1

Получение тензора из растрового изображения,
выполнение шагов TorchVision ToTensor
(преобразование в тензор с элементами
от 0 до 1) и Normalize

@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE &&
        resultCode == RESULT_OK) {
        Bitmap bitmap = (Bitmap) data.getExtras().get("data");

        final float[] means = {0.0f, 0.0f, 0.0f};
        final float[] stds = {1.0f, 1.0f, 1.0f};

        final Tensor inputTensor = TensorImageUtils.bitmapToFloat32Tensor(
            bitmap, means, stds);

        final Tensor outputTensor = model.forward(
            IValue.from(inputTensor)).toTensor();
        Bitmap output_bitmap = tensorToBitmap(outputTensor, means, stds,
            Bitmap.Config.RGB_565);
        image_view.setImageBitmap(output_bitmap);
    }
}

```

← Это выполняется, когда приложение камеры делает фото

← Это выглядит почти как то, что мы делали в C++

← Функция tensorToBitmap — наше собственное изобретение

Вокруг фактического вызова `model.forward` мы реализуем запаковку и распаковку `IValue`, как уже делали при использовании JIT в C++, за исключением того, что метод `forward` берет один `IValue`, а не вектор. Наконец, нам нужно вернуться к растровому изображению. Здесь PyTorch нам не поможет, поэтому нужно определить функцию `tensorToBitmap` (и отправить pull request в PyTorch). Мы избавим вас от скучных подробностей — в них ничего интересного и много копирования (из тензора в массив `float[]`, затем в массив `int[]`, содержащий значения ARGB, затем в растровое изображение), но что уж тут поделать. Эта функция обратна `bitmapToFloat32Tensor`.

Вот и все — мы добавили PyTorch в Android. Добавив еще пару минимальных манипуляций с кодом для запроса изображения, мы получаем Android-приложение Zebraify, похожее на то, что приведено на рис. 15.5. Замечательно!¹

¹ На момент написания библиотека PyTorch Mobile была относительно молода, и вы можете столкнуться с трудностями. В Pytorch 1.3 во время работы в эмуляторе на реальном 32-битном телефоне ARM не работали цвета. Причина, вероятно, заключается в ошибке в одной из вычислительных внутренних функций, которые используются только в ARM. В PyTorch 1.4 и более новых телефонах (64-битная версия ARM) все работает лучше.



Рис. 15.5. Приложение CycleGAN Zebra

Следует отметить, что на Android у нас получилась полная версия PyTorch со всеми операциями. Сюда также входят операции, которые для данной задачи вам не понадобятся, и возникает вопрос: можем ли мы сэкономить место, опустив их. Оказывается, начиная с PyTorch 1.4, вы можете создать собственную версию библиотеки PyTorch, включающую только те операции, которые вам нужны (см. <https://pytorch.org/mobile/android/#custom-build>).

15.5.1. Повышение эффективности: проектирование моделей и квантование

Если мы хотим более плотно поработать с мобильными устройствами, то наш следующий шаг — попытаться сделать модели быстрее. Когда нужно уменьшить объем памяти и объем вычислений модели, первое, на что нужно обратить внимание, — это оптимизация самой модели: то есть вычисление одинаковых или очень похожих преобразований от входных данных к выходным с меньшим количеством параметров и операций. Это часто называют *дистилляцией*. Подходы к дистилляции бывают разные. Иногда уменьшают значения весов,

удаляя маленькие или ненужные веса¹, иногда объединяют несколько слоев сети в один (DistilBERT) или даже обучают другую, более простую модель, которая учится воспроизводить выходные данные более крупной модели (оригинальный CTranslate от OpenNMT). Подобные модификации, вероятно, могут быть первым шагом к ускорению работы моделей.

Другой подход заключается в уменьшении размера каждого параметра и операции: вместо того, чтобы тратить на числа с плавающей запятой 32 бита, мы перенастраиваем модель на работу с целыми числами (обычно восьмибитными). Это называется *квантованием*².

В PyTorch для этого предусмотрены квантованные тензоры. Они представлены как набор скалярных типов, подобных `torch.float`, `torch.double` и `torch.long` (см. раздел 3.5). Наиболее распространенными типами квантованных тензоров являются `torch.quint8` и `torch.qint8`, где числа представляются как восьмибитные целые числа без знака и со знаком соответственно. Для механизма диспетчеризации в PyTorch используется отдельный скалярный тип, который мы кратко рассмотрели в разделе 3.11.

Может показаться удивительным, что использование восьмибитных целых чисел вместо 32-битных чисел с плавающей запятой вообще работает. Да, обычно наблюдается небольшое ухудшение результатов, но незначительное. По-видимому, этому способствуют две вещи: если мы рассматриваем ошибки округления как, по существу, случайные, а свертки и линейные слои как средневзвешенные значения, то ошибки округления обычно компенсируются, а для этого хватает семи бит целого числа со знаком³. Еще одна особенность квантования (в отличие от обучения с 16-битными числами с плавающей запятой) — переход с плавающей на фиксированную точность. Это означает, что самые большие значения разрешаются с точностью до семи бит, а значения, составляющие одну восьмую от самых больших значений, разрешаются только до $7 - 3 = 4$ бит. Но если работают инструменты наподобие регуляризации L1 (кратко упомянутой в главе 8), то мы можем надеяться, что подобные эффекты приводят к снижению точности в определении малых весов. Зачастую так и получается.

Квантование впервые появилось в PyTorch 1.3 и все еще немного грубовато с точки зрения поддерживаемых операций в PyTorch 1.4. Тем не менее библиотека

¹ Например, Lottery Ticket Hypothesis и WaveRNN.

² В отличие от квантования (частичный) переход к 16-битным числам с плавающей запятой для обучения обычно называется редуцированным или (если некоторые биты остаются 32-битными) обучением со смешанной точностью.

³ Пижоны в подобных случаях апеллируют к теореме о центральном пределе. И действительно, следует подумать о сохранении независимости (в статистическом смысле) ошибок округления. Например, обычно требуется, чтобы ноль (важный результат ReLU) был точно представим. В противном случае все нули при округлении заменятся на одно и то же число, что приведет к снежному кому ошибок.

быстро совершенствуется, и мы рекомендуем обратить внимание на эти инструменты, если вы серьезно относитесь к развертыванию в среде, где требуется эффективность.

15.6. НОВЫЕ ТЕХНОЛОГИИ: КОРПОРАТИВНАЯ ПОСТАВКА МОДЕЛЕЙ PYTORCH

Мы можем спросить себя: должны ли все аспекты развертывания, которые мы обсудили, вынуждать писать столько кода. Конечно, достаточно часто кто-то пишет весь код вручную. По состоянию на начало 2020 года, пока мы дописываем книгу, у нас большие надежды на ближайшее будущее и есть ощущение, что уже к лету ландшафт развертывания значительно изменится.

В настоящее время в RedisAI (<https://github.com/RedisAI/redisai-py>), которым занимается один из авторов, уже хотят применить Redis к нашим моделям. PyTorch в экспериментальном порядке выпустили TorchServe (<https://pytorch.org/blog/pytorch-library-updates-new-model-serving-library/#torchserve-experimental>).

MLflow (<https://mlflow.org>) тоже расширяет поддержку, а Cortex (<https://cortex.dev>) хочет, чтобы мы попробовали именно его для развертывания моделей. Для более конкретной задачи поиска информации также есть EuclidesDB (<https://euclidesdb.readthedocs.io/en/latest>) для создания баз данных объектов на основе ИИ.

Грядут интересные времена, но, к сожалению, они не совпадают с нашим графиком написания. Мы надеемся, что во втором издании (или во второй книге) нам будет что рассказать!

15.7. ИТОГИ ГЛАВЫ

На этом мы завершаем наш краткий обзор того, как поместить модель туда, где ей предстоит работать. Хотя готовая версия Torch еще не готова, когда она появится, вы, вероятно, захотите экспортировать свои модели через JIT, поэтому полученные здесь знания окажутся полезны. Вы теперь знаете, как развернуть модель в сетевом сервисе, в приложении C++ или на мобильном устройстве. Мы с нетерпением ждем ваших достижений!

Мы надеемся, что выполнили обещание, данное в этой книге: предоставить вам практические знания основ глубокого обучения и познакомить с библиотекой PyTorch. Мы надеемся, вам понравилось читать так же, как нам понравилось писать¹.

¹ На самом деле писать книги довольно трудно!

15.8. УПРАЖНЕНИЕ

Заканчивая книгу, предлагаем вам последнее упражнение: выберите интересный проект наподобие Kaggle и погрузитесь в него.

Вы приобрели навыки и изучили инструменты, необходимые для достижения успеха. Нам не терпится услышать, что вы будете делать дальше; напишите нам на форуме книги и расскажите!

15.9. РЕЗЮМЕ

- Мы можем поставлять модели PyTorch, поместив их в фреймворк веб-сервера Python, такой как Flask.
- С помощью JIT-моделей мы можем обойти GIL даже при вызове из Python, что полезно для задач поставки моделей.
- Пакетная и асинхронная обработка запросов помогают эффективно использовать ресурсы, особенно когда логический вывод выполняется на графическом процессоре.
- Для экспорта моделей за пределы PyTorch отлично подходит формат ONNX. ONNX Runtime предоставляет серверную часть для многих платформ, включая Raspberry Pi.
- JIT позволяет без особых усилий экспортировать и запускать произвольный код PyTorch на C++ или на мобильных устройствах.
- Отслеживание — самый простой способ получить JIT-модель. Для особо динамичных частей, возможно, лучше будет использовать сценарии.
- У PyTorch также имеется хорошая поддержка C++ (и расширяется поддержка других языков) для запуска моделей как через JIT, так и в исходном коде.
- PyTorch Mobile позволяют легко интегрировать JIT-модели в приложения для Android или iOS.
- При развертывании на мобильном устройстве нужно упростить архитектуру модели и, если возможно, квантовать ее.
- Появляются новые среды развертывания, но стандарт пока не совсем очевиден.

Эли Стивенс, Лука Антига, Томас Виман
PyTorch. Освещая глубокое обучение

Перевели с английского И. Пальти, С. Черников

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературные редакторы	<i>Ю. Зорина, Н. Хлебина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>С. Беляева, Е. Павлович</i>
Верстка	<i>Л. Егорова</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги
печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 24.06.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 46,440. Тираж 800. Заказ 0000.

Лаура Грессер, Ван Лун Кенг

ГЛУБОКОЕ ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ: ТЕОРИЯ И ПРАКТИКА НА ЯЗЫКЕ PYTHON



Глубокое обучение с подкреплением (глубокое RL) сочетает в себе два подхода к машинному обучению. В ходе такого обучения виртуальные агенты учатся решать последовательные задачи о принятии решений. За последнее десятилетие было много неординарных достижений в этой области — от однопользовательских и многопользовательских игр, таких как го и видеоигры Atari и Dota 2, до робототехники.

Эта книга — введение в глубокое обучение с подкреплением, уникально комбинирующее теорию и практику. Авторы начинают повествование с базовых сведений, затем подробно объясняют теорию алгоритмов глубокого RL, демонстрируют их реализации на примере программной библиотеки SLM Lab и напоследок описывают практические аспекты использования глубокого RL.

Руководство идеально подойдет как для студентов, изучающих компьютерные науки, так и для разработчиков программного обеспечения, которые знакомы с основными принципами машинного обучения и знают Python.

КУПИТЬ

Кристиан Майер

ОДНОСТРОЧНИКИ PYTHON: ЛАКОНИЧНЫЙ И СОДЕРЖАТЕЛЬНЫЙ КОД



Краткость — сестра программиста. Эта книга научит вас читать и писать лаконичные и функциональные однострочники. Вы сможете системно разбирать и понимать код на Python, а также писать выразительно и компактно, как настоящий эксперт.

Здесь вы найдете приемы и хитрости написания кода, регулярные выражения, примеры использования однострочников в различных сферах, а также полезные алгоритмы. Подробные пояснения касаются в том числе и важнейших понятий computer science, что поспособствует вашему росту в программировании и аналитике.

КУПИТЬ

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