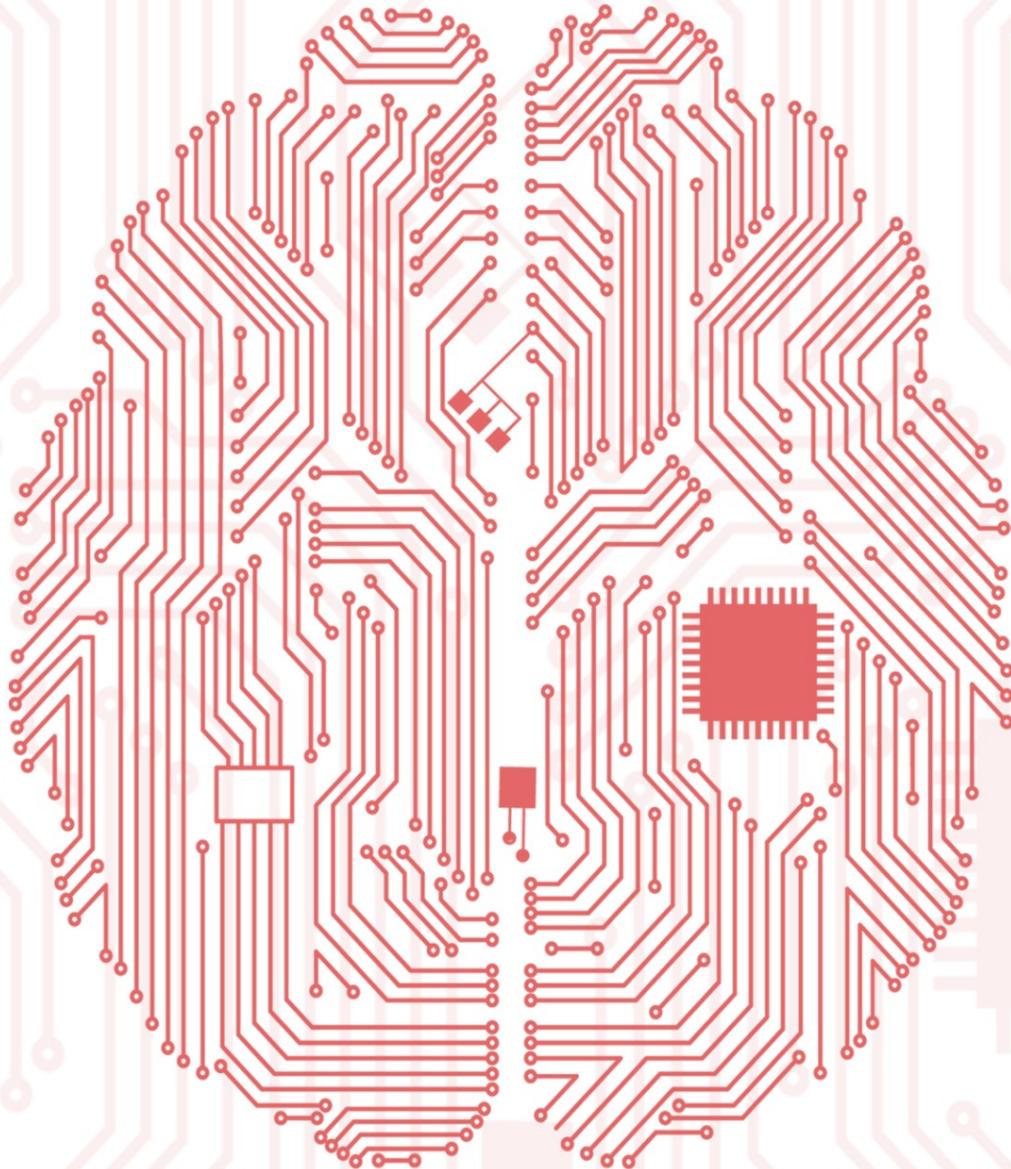


DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

Dr. Adrian Rosebrock

 PyImageSearch

Глубокое обучение для компьютерного зрения с ПИТОН

Стартовый набор

Доктор Адриан Роузброк

1-е издание (1.1.0)

Copyright с 2017 Адриан Роузброк, PyImageSearch.com

ИЗДАНО PYIMAGESearch

PYIMAGESearch.COM

Содержание этой книги, если не указано иное, защищено авторским правом с Adrian Rosebrock, 2017, PyImageSearch.com. Все права защищены. Книги, подобные этой, стали возможными благодаря времени, вложенному авторами. Если вы получили эту книгу, но не купили ее, рассмотрите возможность создания будущих книг , купив копию на [https://www.pyimagesearch.com/deep-learning-computer-vision python-book/](https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/) сегодня.

Первый тираж, сентябрь 2017 г.

Моему отцу Джо; моя жена Триша;
и семейные гончие, Джози и Джемма.
Без их постоянной любви и поддержки
эта книга была бы невозможна.

Содержание

1 Введение .	15
1.1 Я изучал глубокое обучение неправильным путем... Это правильный путь 1.2. Для кого предназначена эта книга 1.2.1. Вы только начинаете заниматься глубоким обучением? .	15 17
1.2.2 Уже являетесь опытным специалистом по глубокому обучению? .	17
1.3 Организация книги	17
1.3.1 Том №1: Стартовый набор.	17
1.3.2 Том № 2: Комплект для практикующих .	18
1.3.3 Том № 3: Пакет ImageNet .	18
1.3.4 Нужно обновить пакет? .	18
1.4 Инструменты торговли: Python, Keras и Mxnet	18
1.4.1 Что насчет TensorFlow? .	18
1.4.2 Нужно ли мне знать OpenCV? .	19
1.5 Разработка собственного набора инструментов для глубокого	19
 обучения 1.6 Резюме	20
2 Что такое глубокое обучение? .	21
2.1 Краткая история нейронных сетей и глубокого обучения 2.2	22
Иерархическое функциональное	24
обучение 2.3 Насколько «глубоко» глубоко?	27
2.4 Резюме	30
3 Основы изображения . 3.1	31
Пиксели изображения блоки	31
изображения из каналов .	34

3.2 Система координат изображения 3.2.1	34
Изображения как массивы NumPy	35
3.2.2 Заказ RGB и BGR	36
3.3 Масштабирование и соотношение	36
сторон 3.4 Резюме	38
4 Основы классификации изображений	39
4.1 Что такое классификация изображений?	40
4.1.1 Примечание по терминологии.	40
4.1.2 Семантический разрыв.	41
4.1.3 Проблемы.	42
4.2 Типы обучения 4.2.1	45
Обучение под наблюдением	45
4.2.2 Обучение без учителя.	46
4.2.3 Полуконтролируемое обучение.	47
4.3 Конвейер классификации глубокого обучения	48
4.3.1 Изменение мышления.	48
4.3.2 Шаг № 1: Соберите свой набор данных.	50
4.3.3 Шаг № 2: Разделите набор данных.	50
4.3.4 Шаг № 3: Обучите свою сеть.	51
4.3.5 Шаг № 4: Оцените	51
4.3.6 Обучение на основе признаков по сравнению с глубоким обучением для классификации изображений.	51
4.3.7 Что происходит, когда мои прогнозы неверны?	52
4.4 Резюме	52
5 Наборы данных для классификации изображений	53
5.1 MNIST	53
5.2 Животные: собаки, кошки и панды 5.3	54
CIFAR-10	55
5.4 УЛЫБКИ	55
5.5 Kaggle: собаки против кошек	56
5.6 Цветы-17	56
5.7 CALTECH-101	57
5.8 Крошечный ImageNet	57
200 5.9 Адиенс	58
5.10 ImageNet 5.10.1	58
Что такое ImageNet?	58
5.10.2 Крупномасштабная программа визуального распознавания ImageNet (ILSVRC)	58
5.11 Kaggle: вызов распознавания лиц 5.12 CVPR в помещении	59
60	60
5.13 Стенфордские автомобили	60
5.14 Резюме	60

6	Настройка среды разработки	63
6.1	Библиотеки и пакеты 6.1.1	63
Python.	63
6.1.2 Керас.	64
6.1.3 Мкснет.	64
6.1.4 OpenCV, scikit-image, scikit-learn и другие.	64
6.2 Настройка среды разработки?	64
6.3 Предварительно настроенная виртуальная машина	65
6.4 Облачные экземпляры	65
6.5 Как структурировать ваши проекты	65
6.6 Резюме	66
7	Ваш первый классификатор изображений	67
7.1	Работа с наборами данных изображений	67
7.1.1 Знакомство с набором данных «Животные».	67
7.1.2 Начало работы с нашим набором инструментов для глубокого обучения.	68
7.1.3 Базовый препроцессор изображений.	69
7.1.4 Создание загрузчика изображений . 7.2 k-NN:	70
простой классификатор 7.2.1 Рабочий пример k-NN. 7.2.2	72
Гиперпараметры k-NN	74
7.2.3 Реализация k-NN. 7.2.4 Результаты k-NN.	75
7.2.5 Плюсы и минусы k-NN.	79
7.3 Резюме	80
8	Параметризованное обучение	81
8.1	Введение в линейную классификацию 8.1.1	82
Четыре компонента параметризованного обучения.	82
8.1.2 Линейная классификация: от изображений к меткам.	83
8.1.3 Преимущества параметризованного обучения и линейной классификации.	84
8.1.4 Простой линейный классификатор с Python.	85
8.2 Роль функций потерь	88
8.2.1 Что такое функции потерь?	88
8.2.2 Потеря мультиклассового SVM	89
8.2.3 Кросс-энтропийные потери и классификаторы Softmax.	91
8.3 Резюме	94
9	Методы оптимизации и регуляризация	95
9.1	Градиентный спуск	96
9.1.1 Ландшафт потерь и поверхность оптимизации.	96
9.1.2 «Градиент» в градиентном спуске.	97
9.1.3 Рассматривайте это как выпуклую задачу (даже если это не так).	98
9.1.4 Уловка предвзятости.	98
9.1.5 Псевдокод для градиентного спуска.	99
9.1.6 Реализация базового градиентного спуска в Python.	100

9.1.7 Результаты простого градиентного спуска	104
9.2 Стохастический градиентный спуск (SGD)	106
9.2.1 Мини-пачка SGD.	106
9.2.2 Реализация мини-пакета SGD.	107
9.2.3 Результаты SGD.	110
9.3 Расширения до SGD	111
9.3.1 Импульс.	111
9.3.2 Ускорение Нестерова.	112
9.3.3 Отдельные рекомендации.	113
9.4 Регуляризация	113
9.4.1 Что такое регуляризация и зачем она нам нужна?	113
9.4.2 Обновление нашего обновления по потерям и весу для включения регуляризации.	115
9.4.3 Типы методов регуляризации.	116
9.4.4 Регуляризация, применяемая к классификации изображений.	117
9.5 Резюме	119
10 Основы нейронных сетей	121
10.1 Основы нейронной сети	121
10.1.1 Введение в нейронные сети.	122
10.1.2 Алгоритм персептрана.	129
10.1.3 Обратное распространение и многоуровневые сети.	137
10.1.4 Многоуровневые сети с Keras.	153
10.1.5 Четыре компонента рецепта нейронной сети.	163
10.1.6 Инициализация веса	165
10.1.7 Инициализация констант	165
10.1.8 Равномерное и нормальное распределения.	165
10.1.9 Равномерное и нормальное по Лекуну.	166
10.1.10 Глорот/Ксавье Униформа и Обычная	166
10.1.11 He et al./Kaiming/MSRA Uniform and Normal	167
10.1.12 Различия в реализации инициализации	167
10.2 Резюме	168
11 Сверточные нейронные сети	169
11.1 Понимание сверток	170
11.1.1 свертки и взаимная корреляция.	170
11.1.2. Аналогия «большой матрицы» и «крошечной матрицы».	171
11.1.3 Ядра	171
11.1.4. Пример ручного вычисления свертки.	172
11.1.5 Реализация сверток с помощью Python.	173
11.1.6 Роль сверток в глубоком обучении.	179
11.2 Строительные блоки CNN	179
11.2.1 Типы слоев	181
11.2.2 Сверточные слои.	181
11.2.3 Уровни активации	186
11.2.4 Объединение слоев	186
11.2.5 Полносвязные слои.	188
11.2.6 Нормализация партии	189
11.2.7 Отключение	190

11.3 Общие архитектуры и шаблоны обучения 11.3.1	191
Шаблоны слоев	191
11.3.2 Практические правила.	192
11.4. Инвариантны ли CNN к трансляции, вращению и масштабированию?	194
11.5 Резюме	195
12 Тренировка вашего первого CNN	197
12.1 Конфигурации Keras и преобразование изображений в массивы	197
12.1.1 Понимание файла конфигурации keras.json.	197
12.1.2 Препроцессор изображения в массив.	198
12.2 Мелкая сеть	200
12.2.1 Реализация ShallowNet.	200
12.2.2 ShallowNet на животных.	202
12.2.3 ShallowNet на CIFAR-10.	206
12.3 Резюме	209
13 Сохранение и загрузка ваших моделей	211
13.1 Сериализация модели на диск	211
13.2 Загрузка предварительно обученной модели	214
с диска 13.3 Резюме	217
14 LeNet: Распознавание рукописных цифр	219
14.1 Архитектура LeNet	219
14.2 Внедрение LeNet 14.3	220
LeNet на MNIST	222
14.4 Резюме	227
15 MiniVGGNet: углубляясь в CNN	229
15.1 Семейство сетей VGG 15.1.1	229
Архитектура (мини) VGGNet	230
15.2 Реализация MiniVGGNet 15.3	230
MiniVGGNet на CIFAR-10	234
15.3.1 С пакетной нормализацией.	236
15.3.2 Без пакетной нормализации.	237
15.4 Резюме	238
16 планировщиков скорости обучения	241
16.1 Снижение скорости обучения	241
16.1.1 Стандартный график затухания в Керасе.	242
16.1.2 Ступенчатое затухание.	243
16.1.3 Внедрение пользовательских расписаний скорости обучения в Keras.	244
16.2 Резюме	249

17 Определение недообучения и переобучения	251
17.1 Что такое недообучение и переоснащение?	251
17.1.1 Влияние скорости обучения.	253
17.1.2 Обратите внимание на свои тренировочные кривые.	254
17.1.3 Что делать, если потери при валидации ниже потерь при обучении?	254
17.2 Мониторинг процесса обучения	255
17.2.1 Создание монитора обучения :	255
17.2.2 Обучение присмотрю за детьми	257
17.3 Резюме	260
18 Модели контрольных точек	263
18.1 Контрольные точки Улучшения модели нейронной сети 18.2	263
Контрольные точки только для лучшей нейронной сети 18.3	267
Резюме	269
19 Визуализация сетевых архитектур	271
19.1 Важность визуализации архитектуры 19.1.1	271
Установка graphviz и pydot	272
19.1.2 Визуализация сетей Keras.	272
19.2 Резюме	275
20 готовых CNN для классификации	277
20.1 Современные CNN в Керасе	277
20.1.1 VGG16 и VGG19.	278
20.1.2 Ренет.	279
20.1.3 Начальная версия V3.	280
20.1.4 Восприятие	280
20.1.5 Может ли мы стать меньше?	280
20.2 Классификация изображений с предварительно обученными CNN	281
ImageNet 20.2.1 Результаты классификации	284
20.3 Резюме	286
21 Практический пример: Взлом капчи с помощью CNN	287
21.1 Взлом капчи с помощью CNN 21.1.1	288
Примечание об ответственном раскрытии информации :	288
21.1.2 Структура каталога устройства для разбивания капчи	290
21.1.3 Автоматическая загрузка примеров изображений	291
21.1.4 Аннотирование и создание нашего набора данных.	292
21.1.5 Предварительная обработка цифр	297
21.1.6 Обучение взломщику капчи	299
21.1.7 Тестирование разбивателя капчи	303
21.2 Резюме	305
22 Практический пример: Распознавание улыбки	307
22.1 Набор данных SMILES	307

22.2 Обучение Smile CNN 22.3	308
Запуск Smile CNN в режиме реального времени 22.4 Резюме	313
23 Ваши следующие шаги	319
23.1 Итак, что дальше?	319

Сопутствующий веб-сайт

Спасибо, что приобрели книгу «Глубокое обучение компьютерному зрению с помощью Python»! В дополнение к этой книге я создал сопутствующий веб-сайт, который включает в себя:

- Актуальные инструкции по настройке среды разработки • Инструкции по использованию предварительно настроенной виртуальной машины Ubuntu VirtualBox и
Образ машины Amazon (AMI)
- Дополнительный материал, который мне не удалось поместить в эту книгу .

Кроме того, вы можете использовать функцию «Проблемы» на сопутствующем веб-сайте, чтобы сообщать о любых ошибках, опечатках или проблемах, с которыми вы сталкиваетесь при работе с книгой. Я не ожидаю много проблем; тем не менее, это совершенно новая книга, поэтому я и другие читатели были бы признательны за сообщение о любых проблемах, с которыми вы столкнетесь. Оттуда я могу обновлять книгу и избавляться от ошибок.

Чтобы создать учетную запись на сопутствующем веб-сайте, просто воспользуйтесь
этой ссылкой: <http://pyimg.co/fnkxk> Потратьте секунду, чтобы создать учетную
запись сейчас, чтобы иметь доступ к дополнительным материалам.
пока вы работаете с книгой.

1. Введение

«Секрет успеха в том, чтобы начать». - Марк Твен

Добро пожаловать в Deep Learning для компьютерного зрения с Python. Эта книга — ваш путеводитель по освоению глубокого обучения, применяемого к практическим, реальным задачам компьютерного зрения с использованием языка программирования Python и библиотек Keras + mxnet. В этой книге вы узнаете, как применять глубокое обучение к таким проектам, как классификация изображений, обнаружение объектов, обучение сетей на крупномасштабных наборах данных и многое другое.

Глубокое обучение для компьютерного зрения с помощью Python стремится обеспечить идеальный баланс между теорией, преподаваемой в классе/учебнике, и фактическими практическими знаниями, которые вам понадобятся для достижения успеха в реальном мире.

Для достижения этой цели вы будете учиться практическим прикладным образом, обучая сети на своих собственных пользовательских наборах данных и даже соревнуясь в сложных задачах и соревнованиях по классификации изображений. К тому времени, как вы закончите читать эту книгу, вы будете хорошо подготовлены для применения глубокого обучения в своих собственных проектах. И я не сомневаюсь, что при достаточной практике вы сможете использовать свои недавно полученные знания, чтобы найти работу в области глубокого обучения, стать консультантом / подрядчиком по глубокому обучению для компьютерного зрения или даже начать свое собственное компьютерное зрение. основанная компания, которая использует глубокое обучение.

Так что хватай свой хайлайтер. Найдите удобное место. И позвольте мне помочь вам на вашем пути к мастерству глубокого обучения. Помните, что самый важный шаг — это первый — просто начать.

1.1 Я неправильно изучал глубокое обучение. . . Это правильный путь

Я хочу начать эту книгу с личной истории:

Ближе к концу моей карьеры в аспирантуре (2013-2014) я начал ломать голову над всем этим «глубоким обучением» из-за временной причуды. Я оказался в очень уникальной ситуации. Моя диссертация была (по сути) завершена. Каждый из моих кандидатов наук. члены комитета подписали его. Однако из-за правил университета/кафедры у меня все еще был дополнительный семестр, который мне нужно было «поторчать», прежде чем я смог официально защитить диссертацию и получить высшее образование. Этот разрыв по существу

оставил мне целый семестр (4 месяца) впустую — это было отличное время, чтобы начать изучать глубокое обучение.

Моей первой остановкой, как и большинства ученых, было прочтение всех последних публикаций по глубокому обучению. Благодаря моему опыту машинного обучения мне не потребовалось много времени, чтобы понять фактические теоретические основы глубокого обучения.

Однако я придерживаюсь мнения, что пока вы не возьмете свои теоретические знания и не примените их, вы еще ничему не научитесь. Преобразование теории в реализацию — это совсем другой процесс, как скажет вам любой ученый-компьютерщик, который ранее посещал занятия по структурам данных : чтение о красно-черных деревьях, а затем фактическая реализация их с нуля требует двух разных наборов навыков.

И именно в этом заключалась моя проблема.

После прочтения этих публикаций по глубокому обучению я остался чесать затылок; Я не мог взять то, что узнал из статей, и реализовать настоящие алгоритмы, не говоря уже о том, чтобы воспроизвести результаты.

Разочарованный своими неудачными попытками реализации, я часами искал в Google учебники по глубокому обучению, но нашел их с пустыми руками. В то время было не так много руководств по глубокому обучению.

Наконец, я решил поиграть с библиотеками и инструментами, такими как Caffe, Theano и Torch. слепо следовал плохо написанным сообщениям в блоге (с неоднозначными результатами, если не сказать больше).

Я хотел начать, но на самом деле еще ничего не щелкнуло — лампочка глубокого обучения в моей голове застряла в положении «выключено».

Чтобы быть полностью честным с вами, это был болезненный, эмоционально тяжелый семестр. Я мог ясно видеть ценность глубокого обучения для компьютерного зрения, но мне нечего было показать за свои усилия, кроме стопки документов по глубокому обучению на моем столе, которые я понимал, но изо всех сил пытался реализовать.

В течение последнего месяца семестра я, наконец, нашел свой путь к успеху в глубоком обучении с помощью сотен экспериментов методом проб и ошибок, бесчисленных ночей и большой настойчивости. В конечном итоге эти четыре месяца оказали огромное влияние на мою жизнь, мой исследовательский путь и то, как я сегодня понимаю глубокое обучение... но я бы не советовал вам идти тем же путем, что и я.

...

Если взять что-то из моего личного опыта, то оно должно быть таким: 1. Вам не нужно десятилетие теории, чтобы начать глубокое обучение.

2. Вам не нужны страницы и страницы уравнений.

3. И вам, конечно, не нужна степень в области компьютерных наук (хотя она может быть полезной).

Когда я начал изучать глубокое обучение, я совершил критическую ошибку, глубоко погрузившись в публикации, никогда не возвращаясь к ним, чтобы попытаться реализовать то, что я изучил. Не поймите меня неправильно – теория важна. Но если вы не используете (или не можете) свои недавно полученные теоретические знания и применяете их для создания реальных реальных приложений, вам будет сложно найти свое место в мире глубокого обучения.

Глубокое обучение и большинство других специализированных предметов компьютерных наук более высокого уровня признают, что теоретических знаний недостаточно — нам также необходимо быть практиками в соответствующих областях. На самом деле идея стать практиком глубокого обучения была моей точной мотивацией при написании Deep Learning for Computer Vision with Python.

Пока есть:

1. Учебники, которые научат вас теоретическим основам машинного обучения, нейронных сетей и глубокого обучения. 2. И бесчисленные ресурсы в стиле «поваренной книги», которые «покажут вам в коде», но никогда не свяжут код с истинными теоретическими знаниями... .

... ни одна из этих книг или ресурсов не будет служить мостом между другими.

На одной стороне моста у вас есть учебники, глубоко укоренившиеся в теории и абстракции.

А с другой стороны, у вас есть книги «покажи мне в коде», которые просто представляют вам примеры,

возможно, объясняя код, но никогда не связывая код с лежащей в его основе теорией.

Между этими двумя стилями обучения существует фундаментальный разрыв, разрыв, который я хочу чтобы помочь заполнить, чтобы вы могли учиться лучше и эффективнее.

Я вспомнил дни учебы в аспирантуре, чувства разочарования и раздражения, дни, когда я даже подумывал сдаться. Я направил эти чувства, когда сел писать эту книгу. Книга, которую вы сейчас читаете, — это книга, которую я хотел бы иметь, когда впервые начал изучать глубокое обучение.

В оставшейся части книги «Глубокое обучение компьютерному зрению с помощью Python» вы найдете очень практические пошаговые руководства, практические руководства (с большим количеством кода) и серьезный стиль обучения, который гарантированно избавит вас от всего лишнего и поможет вам освоить глубокое обучение для компьютерного зрения.

Будьте уверены, вы в надежных руках — это именно та книга, которую вы искали, и Я невероятно рад присоединиться к вам в вашем путешествии по глубокому обучению визуальному распознаванию.

1.2 Для кого эта книга

Эта книга предназначена для разработчиков, исследователей и студентов, которые хотят освоить глубокое обучение для компьютерного зрения и визуального распознавания.

1.2.1 Только начинаете работать с глубоким обучением?

Не волнуйся. Вы не увязнете в тоннах теории и сложных уравнений. Мы начнем с основ машинного обучения и нейронных сетей. Вы будете учиться в веселой, практической форме с большим количеством кода. Я также предоставлю вам ссылки на основополагающие статьи в литературе по машинному обучению, которые вы сможете использовать для расширения своих знаний, как только почувствуете, что у вас есть прочная основа, на которую можно опереться.

Самый важный шаг, который вы можете сделать прямо сейчас, — просто начать. Позволь мне позаботиться об обучении — независимо от твоего уровня мастерства, поверь мне, ты не останешься позади. К тому времени, когда вы закончите первые несколько глав этой книги, вы станете ниндзя нейронных сетей и сможете перейти к более сложному содержанию.

1.2.2 Уже являетесь опытным специалистом по глубокому обучению?

Эта книга предназначена не только для начинающих — здесь есть и продвинутый контент. Для каждой главы этой книги я привожу набор академических ссылок, которые вы можете использовать для расширения своих знаний. Многие главы в книге «Глубокое обучение компьютерному зрению с помощью Python» на самом деле объясняют эти академические концепции таким образом, который легко понять и усвоить.

Лучше всего то, что решения и тактики, которые я предлагаю в этой книге, могут быть непосредственно применены к вашей текущей работе и исследованиям. Время, которое вы сэкономите, прочитав «Глубокое обучение компьютерному зрению с помощью Python», с лихвой окупит себя, как только вы примените свои знания к своим проектам/исследованиям.

1.3 Организация книги

Поскольку эта книга охватывает огромное количество материалов, я разбил ее на три тома, которые называются «комплектами». Каждый комплект последовательно строится поверх другого и включает в себя все главы из нижних томов. Вы можете найти краткую разбивку пакетов ниже.

1.3.1 Том №1: Стартовый набор

Starter Bundle отлично подойдет, если вы делаете первые шаги к глубокому обучению для овладения классификацией изображений.

Вы изучите основы:

1. Машинное обучение
2. Нейронные сети
3. Сверточные нейронные сети
4. Как работать с собственными пользовательскими наборами данных

1.3.2 Том № 2: Комплект для практикующих

Пакет Practitioner Bundle основан на Starter Bundle и идеально подходит, если вы хотите углубленно изучать глубокое обучение, разбираться в передовых методах и знакомиться с общими передовыми практиками и эмпирическими правилами.

1.3.3 Том № 3: Пакет ImageNet Пакет ImageNet

представляет собой полное средство глубокого обучения для работы с компьютерным зрением. В этом томе книги я показываю, как обучать крупномасштабные нейронные сети на огромном наборе данных ImageNet , а также анализировать реальные примеры, включая предсказание возраста и пола, идентификацию марки и модели автомобиля, распознавание выражения лица и многое другое . более.

1.3.4 Нужно обновить пакет?

Если вы когда-нибудь захотите обновить свой пакет, все, что вам нужно сделать, это отправить мне сообщение, и мы позаботимся об обновлении как можно скорее:

<http://www.pyimagesearch.com/contact/>

1.4 Торговые инструменты: Python, Keras и Mxnet

Мы будем использовать язык программирования Python для всех примеров в этой книге. Python — чрезвычайно простой язык для изучения. Имеет интуитивно понятный синтаксис. Является супер мощным. И это лучший способ работы с алгоритмами глубокого обучения.

Основной библиотекой глубокого обучения, которую мы будем использовать, является Keras [1]. Библиотека Keras поддерживается блестящим Франсуа Шолле, исследователем глубокого обучения и инженером Google. Я использую Keras уже много лет и могу сказать, что это мой любимый пакет глубокого обучения. Как минимальная модульная сетевая библиотека, которая может использовать Theano или TensorFlow в качестве серверной части, вы просто не можете превзойти Keras.

Вторая библиотека глубокого обучения, которую мы будем использовать, — это mxnet [2] (только ImageNet Bundle), легкая, переносимая и гибкая библиотека глубокого обучения. Пакет mxnet обеспечивает привязку к языку программирования Python и специализируется на распределенном многомашинном обучении — возможность распараллелить обучение между графическими процессорами/устройствами/узлами имеет решающее значение при обучении архитектур глубоких нейронных сетей на массивных наборах данных (таких как ImageNet).

Наконец, мы также будем использовать несколько библиотек компьютерного зрения, обработки изображений и машинного обучения, таких как OpenCV, scikit-image, scikit-learn и т. д.

Python, Keras и mxnet — это хорошо продуманные инструменты, которые при более тесном сочетании создают мощную среду разработки для глубокого обучения, которую можно использовать для освоения глубокого обучения для визуального распознавания.

1.4.1 Что насчет TensorFlow?

TensorFlow [3] и Theano [4] — библиотеки для определения абстрактных графов вычислений общего назначения . Хотя они используются для глубокого обучения, они не являются фреймворками глубокого обучения и на самом деле используются для очень многих других приложений, помимо глубокого обучения.

Keras, с другой стороны, представляет собой среду глубокого обучения, которая предоставляет хорошо разработанный API, облегчающий создание глубоких нейронных сетей. Под капотом Keras использует либо

Вычислительный бэкенд TensorFlow или Theano, позволяющий использовать преимущества этих мощных вычислительных механизмов.

Думайте о вычислительном бэкенде как о двигателе, который работает в вашей машине. Вы можете заменить детали в своем двигателе, оптимизировать другие или полностью заменить двигатель (конечно, при условии, что двигатель соответствует набору спецификаций). Использование Keras обеспечивает переносимость между движками и возможность выбрать лучший движок для вашего проекта.

Думая об этом под другим углом, используя TensorFlow или Theano для построения глубокой нейронной сети. было бы похоже на использование исключительно NumPy для создания классификатора машинного обучения.

Является ли это возможным? Абсолютно.

Однако более выгодно использовать библиотеку, предназначенную для машинного обучения, такую как scikit-learn [5], а не изобретать велосипед с помощью NumPy (и за счет на порядок большего количества кода).

В том же духе Keras находится поверх TensorFlow или Theano, наслаждаясь: 1. Преимуществами мощного базового вычислительного механизма 2. API, упрощающим создание собственных сетей глубокого обучения. Кроме того, поскольку Keras будет добавлен к основной библиотеке TensorFlow в Google [6], мы всегда можем интегрировать код TensorFlow непосредственно в наши модели Keras, если мы того пожелаем. Во многих отношениях мы получаем лучшее из обоих миров, используя Keras.

1.4.2 Нужно ли мне знать OpenCV?

Вам не нужно знать библиотеку компьютерного зрения и обработки изображений OpenCV [7], чтобы успешно читать эту книгу.

Мы используем OpenCV только для облегчения основных операций обработки изображений, таких как загрузка изображения с диска, отображение его на нашем экране и несколько других основных операций.

Тем не менее, небольшой опыт работы с OpenCV имеет большое значение, поэтому, если вы новичок в OpenCV и компьютерном зрении, я настоятельно рекомендую вам изучить эту книгу и другую мою публикацию, «Практический Python и OpenCV» [8] в tandemе.

Помните, что глубокое обучение — это только одна грань компьютерного зрения.
методы видения, которые вы должны изучить, чтобы завершить свои знания.

1.5 Разработка собственного набора инструментов для глубокого обучения

Одним из источников вдохновения для написания этой книги была демонстрация использования существующих библиотек глубокого обучения для создания нашего собственного пользовательского набора инструментов на основе Python, позволяющего нам обучать наши собственные сети глубокого обучения.

Однако эта книга — не просто набор инструментов для глубокого обучения. . . . этот набор инструментов точно такой же, как у меня разрабатывал и совершенствовал за последние несколько лет, занимаясь исследованиями и разработками в области глубокого обучения.

По мере изучения этой книги мы будем создавать компоненты этого набора инструментов по одному. Посредством По окончании Deep Learning for Computer Vision with Python наш набор инструментов сможет:

1. Загрузите наборы данных изображений с диска, сохраните их в памяти или запишите в оптимизированную базу данных. формат.
2. Предварительно обработайте изображения, чтобы они подходили для обучения сверточной нейронной сети.
3. Создайте класс чертежа, который можно использовать для создания собственных пользовательских реализаций Сверточные нейронные сети.
4. Внедрить вручную популярные архитектуры CNN, такие как AlexNet, VGGNet, GoogLeNet, ResNet и SqueezeNet (и обучать их с нуля). 5. . . и многое другое!

1.6 Резюме

Мы живем в особое время машинного обучения, нейронных сетей и истории глубокого обучения. Никогда еще в истории машинного обучения и нейронных сетей доступные инструменты не были такими исключительными.

С точки зрения программного обеспечения у нас есть такие библиотеки, как Keras и mxnet, в комплекте с привязками Python, что позволяет нам быстро создавать архитектуры глубокого обучения за долю времени, которое у нас занимало несколько лет назад.

Затем, с точки зрения аппаратного обеспечения, графические процессоры общего назначения становятся все более дешевыми, но при этом становятся все более мощными. Одни только достижения в технологии графических процессоров позволили любому человеку со скромным бюджетом построить даже простую игровую установку для проведения значимых исследований в области глубокого обучения.

Глубокое обучение — захватывающая область, и благодаря этим мощным графическим процессорам, модульным библиотекам и безудержным интеллектуальным исследователям мы ежемесячно видим новые публикации, продвигающие самые современные достижения.

Видите ли, сейчас самое время заняться изучением глубокого обучения для компьютерного зрения.

Не упустите этот момент в истории — вы не только должны быть частью глубокого обучения, но и те, кто рано извлечет выгоду, обязательно увидят огромную отдачу от своих вложений времени, ресурсов и творчества.

Наслаждайтесь этой книгой. Я рад видеть, куда это приведет вас в этой удивительной области.

2. Что такое глубокое обучение?

«Методы глубокого обучения — это методы репрезентативного обучения с несколькими уровнями представления, полученные путем составления простых, но нелинейных модулей, каждый из которых преобразует представление на одном уровне (начиная с необработанных входных данных) в представление на более высоком, немного более абстрактном уровне. [...] Ключевым аспектом глубокого обучения является то, что эти уровни не разрабатываются инженерами-людьми: они изучаются из данных с использованием процедуры обучения общего назначения», — Янн Лекун, Йошуа Бенджио и Джонатан Хинтон, Nature 2015. [9]

Глубокое обучение — это подполе машинного обучения, которое, в свою очередь, является подполем искусственного интеллекта. интеллект (ИИ). Графическое изображение этой взаимосвязи см. на рис. 2.1.

Основная цель ИИ — предоставить набор алгоритмов и методов, которые можно использовать для решения задач, которые люди решают интуитивно и почти автоматически, но в остальном они очень сложны для компьютеров. Прекрасным примером такого класса проблем ИИ является интерпретация и понимание содержимого изображения — эта задача может быть решена человеком практически без усилий, но оказалось, что машинам выполнить ее чрезвычайно сложно.

В то время как ИИ воплощает в себе большой и разнообразный набор работ, связанных с автоматическим машинным мышлением (логический вывод, планирование, эвристика и т. д.), под область машинного обучения, как правило, особенно заинтересована в распознавании образов и обучении на основе данных.

Искусственные нейронные сети (ИНС) — это класс алгоритмов машинного обучения, которые учатся на основе данных и специализируются на распознавании образов, вдохновленных структурой и функциями мозга. Как мы увидим, глубокое обучение относится к семейству алгоритмов ИНС, и в большинстве случаев эти два термина можно использовать взаимозаменяя. На самом деле, вы можете быть удивлены, узнав, что область глубокого обучения существует уже более 60 лет, она имеет разные названия и воплощения в зависимости от тенденций исследований, доступного оборудования и наборов данных, а также популярных вариантов выдающихся исследователей в то время.

В оставшейся части этой главы мы рассмотрим краткую историю глубокого обучения, обсудим, что делает нейронную сеть «глубокой», и откроем для себя концепцию «иерархического обучения» и то, как оно сделало глубокое обучение одной из главных историй успеха. в современном машинном обучении и компьютерном зрении.

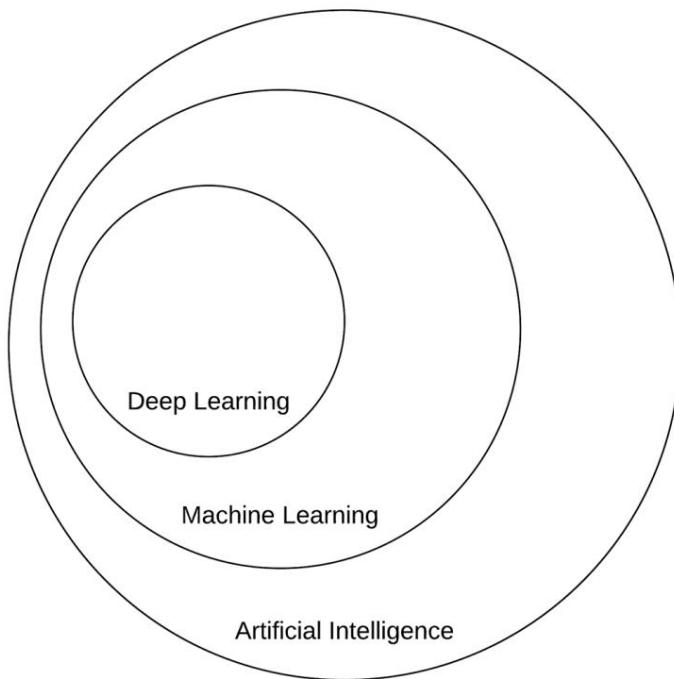


Рисунок 2.1: Диаграмма Венна, описывающая глубокое обучение как подполе машинного обучения, которое, в свою очередь, является подполем искусственного интеллекта (изображение, вдохновленное рис. 1.4 Гудфеллоу и др. [10]).

2.1 Краткая история нейронных сетей и глубокого обучения

История нейронных сетей и глубокого обучения длинная и несколько запутанная. Вас может удивить тот факт, что «глубокое обучение» существует с 1940-х годов, претерпевая различные изменения названий, включая кибернетику, коннекционизм и наиболее знакомые — искусственные нейронные сети (ИНС).

Хотя ИНС вдохновлены человеческим мозгом и тем, как его нейроны взаимодействуют друг с другом, они не предназначены для того, чтобы быть реалистичными моделями мозга. Наоборот, они вдохновляют нас, позволяя проводить параллели между очень простой моделью мозга и тем, как мы можем имитировать некоторые из этих действий с помощью искусственных нейронных сетей. Мы обсудим ИНС и их связь с мозгом в главе 10.

Первая модель нейронной сети была разработана Маккаллохом и Питтсом в 1943 году [11]. Эта сеть представляла собой двоичный классификатор, способный распознавать две разные категории на основе некоторых входных данных. Проблема заключалась в том, что веса, используемые для определения метки класса для данного ввода, должны были быть вручную настроены человеком — этот тип модели явно плохо масштабируется, если требуется вмешательство человека-оператора.

Затем, в 1950-х годах, Розенблatt опубликовал основополагающий алгоритм Perceptron [12, 13] — эта модель могла автоматически определять веса, необходимые для классификации входных данных (вмешательство человека не требовалось). Пример архитектуры персептрона можно увидеть на рис. 2.2. Фактически, эта автоматическая процедура обучения легла в основу стохастического градиентного спуска (SGD), который до сих пор используется для обучения очень глубоких нейронных сетей.

В то время методы, основанные на персептроне, были в моде в сообществе нейронных сетей. Однако публикация Мински и Пейперта в 1969 году [14] практически затормозила исследования нейронных сетей почти на десятилетие. Их работа продемонстрировала, что персептрон с линейной функцией активации (независимо от глубины) был просто линейным классификатором, неспособным решать нелинейные задачи. Каноническим примером нелинейной задачи является набор данных XOR на рис. 2.3. Потратите секунду, чтобы убедить себя в том, что невозможно провести одну линию, которая могла бы разделить синий цвет.

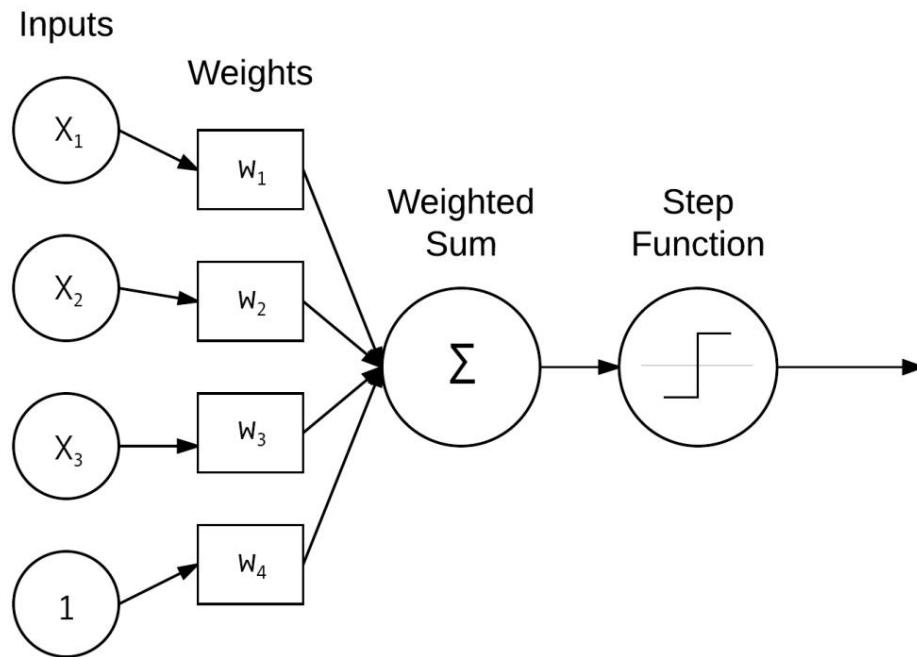


Рисунок 2.2: Пример простой сетевой архитектуры Perceptron, которая принимает ряд входных данных, вычисляет взвешенную сумму и применяет ступенчатую функцию для получения окончательного прогноза. Мы подробно рассмотрим персептрон в главе 10.

звезды из красных кругов.

Кроме того, авторы утверждали, что (в то время) у нас не было вычислительных ресурсов, необходимых для построения больших и глубоких нейронных сетей (оглядываясь назад, они были абсолютно правы). Одна только эта статья чуть не убила исследование нейронных сетей.

К счастью, алгоритм обратного распространения ошибки и исследования Вербоса (1974) [15], Румелхарта (1986) [16] и Лекуна (1998) [17] смогли реанимировать нейронные сети после того, что могло бы стать ранней кончиной. Их исследование алгоритма обратного распространения позволило обучить многослойные нейронные сети с прямой связью (рис. 2.4).

В сочетании с нелинейными функциями активации исследователи теперь могут изучать нелинейные функции и решать проблему исключающего ИЛИ, открывая ворота в совершенно новую область исследований в области нейронных сетей. Дальнейшие исследования показали, что нейронные сети являются универсальными аппроксиматорами [18], способными аппроксимировать любую непрерывную функцию (но не давая гарантии того, может ли сеть действительно изучить параметры, необходимые для представления функции).

Алгоритм обратного распространения является краеугольным камнем современных нейронных сетей, позволяя нам эффективно обучать нейронные сети и «учить» их учиться на своих ошибках. Но даже в этом случае в то время из-за (1) медленных компьютеров (по сравнению с современными машинами) и (2) отсутствия больших помеченных обучающих наборов исследователи не могли (надежно) обучать нейронные сети, которые имели более двух скрытых слоев — это было просто вычислительно невыполнимо.

Сегодня последнее воплощение известных нам нейронных сетей называется глубоким обучением. Что отличает глубокое обучение от его предыдущих воплощений, так это то, что у нас есть более быстрое специализированное оборудование с более доступными обучающими данными. Теперь мы можем обучать сети с гораздо большим количеством скрытых слоев, способных к иерархическому обучению, когда простые понятия изучаются на нижних уровнях, а более абстрактные шаблоны — на более высоких уровнях сети.

Возможно, типичным примером применения глубокого обучения для изучения признаков является Convolutional Neural Network (ConvNet), который был разработан для классификации изображений.

XOR Dataset (Nonlinearly Separable)

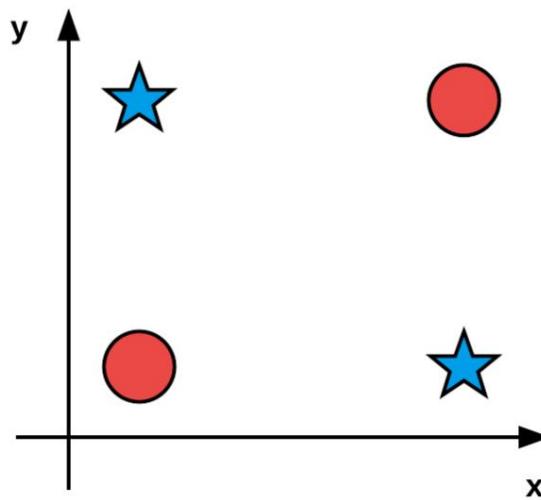


Рисунок 2.3: Набор данных XOR ($(E(X) \text{clive Or})$) является примером нелинейной разделимой задачи, которую персептрон не может решить. Потратьте секунду, чтобы убедить себя, что невозможно провести единую линию, отделяющую синие звезды от красных кругов.

Инновационная нейронная сеть (LeCun, 1988) [19], применяемая для распознавания рукописных символов, которая автоматически изучает отличительные шаблоны (называемые «фильтрами») из изображений путем последовательного наложения слоев друг на друга. Фильтры на более низких уровнях сети представляют края и углы, в то время как слои более высокого уровня используют края и углы для изучения более абстрактных понятий, полезных для различия классов изображений.

Во многих приложениях CNN теперь считаются самым мощным классификатором изображений и в настоящее время отвечают за продвижение современного искусства в подобластях компьютерного зрения, которые используют машинное обучение. Более подробный обзор истории нейронных сетей и глубокого обучения см. в Goodfellow et al. [10], а также отличный пост в блоге Джейсона Браунли из Machine Learning Mastery [20].

2.2 Иерархическое изучение функций

Алгоритмы машинного обучения (как правило) делятся на три лагеря: контролируемое, неконтролируемое и частично контролируемое обучение. В этой главе мы обсудим контролируемое и неконтролируемое обучение, а частично контролируемое обучение оставим для будущего обсуждения.

В контролируемом случае алгоритм машинного обучения получает как набор входных данных, так и целевые выходные данные. Затем алгоритм пытается изучить шаблоны, которые можно использовать для автоматического сопоставления точек входных данных с их правильными целевыми выходными данными. Обучение под наблюдением похоже на то, как учитель наблюдает за тем, как вы сдаете тест. Учитывая ваши предыдущие знания, вы делаете все возможное, чтобы отметить правильный ответ на экзамене; однако, если вы ошиблись, ваш учитель направит вас к более точному и обоснованному предположению в следующий раз.

В неконтролируемом случае алгоритмы машинного обучения пытаются автоматически обнаружить отличительные признаки без каких-либо подсказок относительно входных данных. В этом сценарии наш ученик пытается сгруппировать похожие вопросы и ответы вместе, даже если ученик не знает, какой ответ правильный, а учитель не может дать ему верный ответ. Без присмотра

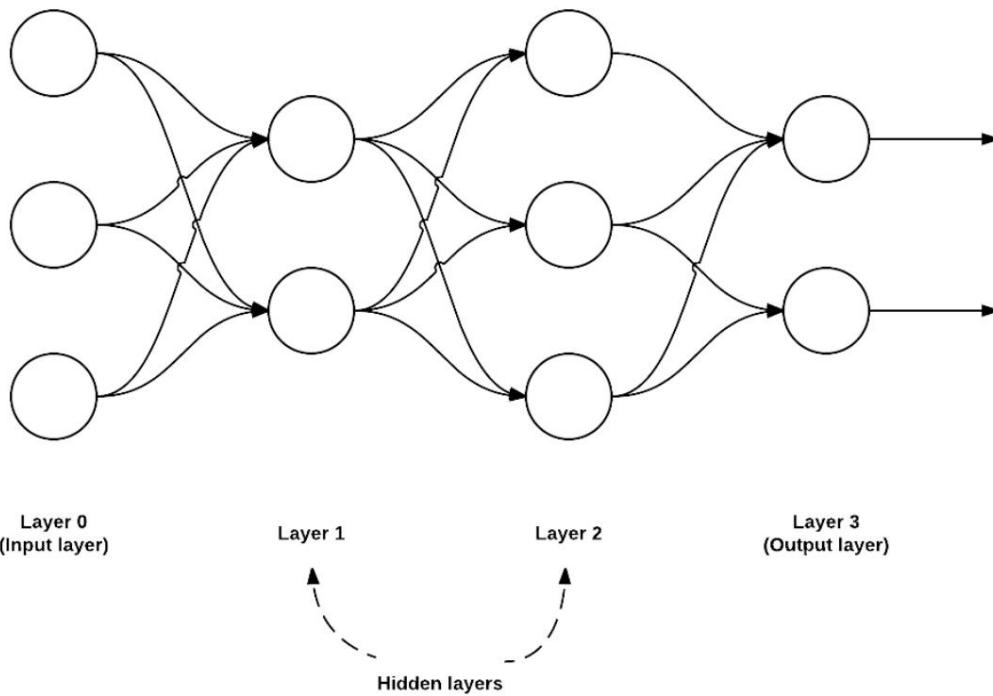


Рисунок 2.4: Многоуровневая сетевая архитектура с прямой связью со входным уровнем (3 узла), двумя скрытыми уровнями (2 узла на первом уровне и 3 узла на втором уровне) и выходным уровнем (2 узла).

обучение, безусловно, является более сложной задачей, чем обучение с учителем: зная ответы (т. е. целевые результаты), мы можем легче определить шаблоны различия, которые могут сопоставить входные данные с правильной целевой классификацией.

В контексте машинного обучения, применяемого к классификации изображений, цель алгоритма машинного обучения состоит в том, чтобы взять эти наборы изображений и идентифицировать шаблоны, которые можно использовать для различия различных классов/объектов изображений друг от друга.

В прошлом мы использовали созданные вручную функции для количественной оценки содержимого изображения — мы редко использовали необработанные интенсивности пикселей в качестве входных данных для наших моделей машинного обучения, как это сейчас принято при глубоком обучении. Для каждого изображения в нашем наборе данных мы выполнили извлечение признаков или процесс получения входного изображения, количественного определения его в соответствии с некоторым алгоритмом (называемым экстрактором признаков или дескриптором изображения) и возвратом вектора (т. е. списка чисел), который предназначен для количественной оценки содержания изображения. На рис. 2.5 ниже показан процесс количественной оценки изображения, содержащего рецептурные таблетки, с помощью ряда дескрипторов изображения цвета, текстуры и формы черного ящика.

Наши созданные вручную функции пытались кодировать текстуру (локальные бинарные паттерны [21], текстура Харалика [22]), форму (моменты Ху [23], моменты Зернике [24]) и цвет (цветовые моменты, цветовые гистограммы, цветовые коррелограммы). [25].

Другие методы, такие как детекторы ключевых точек (FAST [26], Harris [27], DoG [28] и многие другие) и локальные инвариантные дескрипторы (SIFT [28], SURF [29], BRIEF [30], ORB [31].] и т. д.) описывают выделяющиеся (т. е. наиболее «интересные») области изображения.

Другие методы, такие как гистограмма ориентированных градиентов (HOG) [32], оказались очень хорошими при обнаружении объектов на изображениях, когда угол обзора нашего изображения не сильно отличался от того, на котором был обучен наш классификатор. Пример использования метода детектора HOG + Linear SVM



Рисунок 2.5: Количествоное определение содержимого изображения, содержащего рецептурные таблетки, с помощью ряда дескрипторов изображения цвета, текстуры и формы черного ящика.

можно увидеть на рис. 2.6, где мы обнаруживаем наличие знаков остановки на изображениях.

Некоторое время исследования по обнаружению объектов на изображениях ориентировались на HOG и его варианты, включая дорогостоящие в вычислительном отношении методы, такие как модель деформируемых частей [34] и Exemplar SVM [35].

 Для более глубокого изучения дескрипторов изображений, извлечения признаков и процесса, который они играют в компьютерном зрении, обязательно обратитесь к курсу PyImageSearch Gurus [33].

В каждой из этих ситуаций вручную определялся алгоритм для количественной оценки и кодирования определенного аспекта изображения (т. е. формы, текстуры, цвета и т. д.). Получив входное изображение пикселей, мы применили бы наш алгоритм, определенный вручную, к пикселям, а взамен получили бы вектор признаков, количественно определяющий содержимое изображения — сами пиксели изображения не служили никакой другой цели, кроме как входные данные для нашего процесса извлечения признаков. Векторы признаков, полученные в результате извлечения признаков, были тем, что нас действительно интересовало, поскольку они служили входными данными для наших моделей машинного обучения.

Глубокое обучение и, в частности, сверточные нейронные сети используют другой подход. Вместо того, чтобы вручную определять набор правил и алгоритмов для извлечения функций из изображения, эти функции автоматически изучаются в процессе обучения.

Опять же, давайте вернемся к цели машинного обучения: компьютеры должны уметь учиться на опыте (т. е. на примерах) проблемы, которую они пытаются решить.

Используя глубокое обучение, мы пытаемся понять проблему с точки зрения иерархии понятий. Каждая концепция строится поверх других. Концепции на более низких уровнях сети кодируют некоторое базовое представление проблемы, тогда как более высокие уровни используют эти базовые уровни для формирования более абстрактных концепций. Это иерархическое обучение позволяет нам полностью отказаться от ручного процесса извлечения признаков и рассматривать CNN как сквозные обучающиеся.

Учитывая изображение, мы предоставляем значения интенсивности пикселей в качестве входных данных для CNN. Ряд скрытых слоев используется для извлечения функций из нашего входного изображения. Эти скрытые слои строятся друг на друге иерархическим образом. Сначала в нижних слоях сети обнаруживаются только рёберные области. Эти краевые области используются для определения углов (места пересечения краев) и контуров (очертания объектов). Объединение углов и контуров может привести к абстрактным «частям объекта» на следующем слое.

Опять же, имейте в виду, что типы понятий, которые эти фильтры учатся обнаруживать, изучаются автоматически — мы не вмешиваемся в процесс обучения. Наконец, выходной слой



Рисунок 2.6: Инфраструктура обнаружения объектов HOG + Linear SVM, применяемая для определения местоположения знаков остановки на изображениях, как описано в курсе [PyImageSearch Gurus](#) [33].

используется для классификации изображения и получения метки выходного класса — на выходной слой прямо или косвенно влияет каждый другой узел в сети.

Мы можем рассматривать этот процесс как иерархическое обучение: каждый уровень в сети использует выходные данные предыдущих уровней в качестве «строительных блоков» для создания все более абстрактных понятий. Эти слои изучаются автоматически — в нашей сети нет ручной разработки признаков . На рис. 2.7 классические алгоритмы классификации изображений с использованием созданных вручную признаков сравниваются с изучением представлений с помощью глубокого обучения и сверточных нейронных сетей.

Одним из основных преимуществ глубокого обучения и сверточных нейронных сетей является то, что они позволяют нам пропустить этап извлечения признаков и вместо этого сосредоточиться на процессе обучения нашей сети обучению этим фильтрам. Однако, как мы узнаем позже в этой книге, обучение сети для получения разумной точности на заданном наборе данных изображения не всегда является легкой задачей.

2.3 Насколько "глубоко" глубоко?

Цитируя Джекфа Дина из его доклада 2016 года «Глубокое обучение для создания интеллектуальных компьютерных систем» [36]:

«Когда вы слышите термин «глубокое обучение», просто подумайте о большой глубокой нейронной сети. Глубокий обычно относится к количеству слоев, поэтому этот популярный термин был принят в прессе».

Это отличная цитата, поскольку она позволяет нам концептуализировать глубокое обучение как большие нейронные сети, в которых слои строятся друг над другом, постепенно увеличиваясь в глубину. Проблема в том, что у нас до сих пор нет конкретного ответа на вопрос: «Сколько слоев нужно нейронной сети, чтобы считать ее глубокой?»

Короткий ответ: среди экспертов нет единого мнения о том, какой должна быть глубина сети. считается глубоким [10].

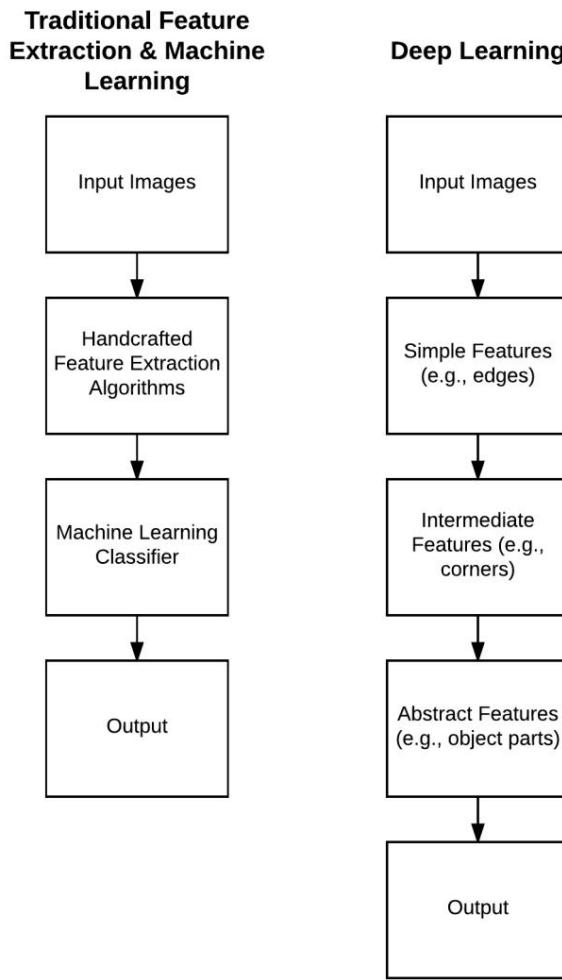


Рисунок 2.7: Слева: традиционный процесс получения входного набора изображений, применения разработанных вручную алгоритмов извлечения признаков с последующим обучением классификатора машинного обучения на этих признаках. Справа: метод глубокого обучения с наложением слоев друг на друга, который автоматически изучает более сложные, абстрактные и отличительные функции.

А теперь нам нужно рассмотреть вопрос о типе сети. По определению сверточная нейронная сеть (CNN) — это тип алгоритма глубокого обучения. Но предположим, что у нас есть CNN только с одним сверточным слоем — является ли сеть, которая является мелкой, но все же принадлежит семейству алгоритмов внутри лагеря глубокого обучения, считается «глубокой»?

Мое личное мнение состоит в том, что любая сеть с более чем двумя скрытыми слоями может считаться "глубокой". Мои рассуждения основаны на предыдущих исследованиях ИНС, которые сильно пострадали из-

за: 1. Отсутствие у нас больших помеченных наборов данных, доступных для обучения 2. Наши компьютеры слишком медленны для обучения больших нейронных сетей 3. Неадекватные функции активации Из-за этих проблем мы не могли легко обучать сети с более чем двумя скрытыми слоями в 1980-х и 1990-х годах (и, конечно, раньше). На самом деле, Джекф Хинтон поддерживает это мнение в своем выступлении 2016 года «Глубокое обучение» [37], где он обсуждал, почему предыдущие воплощения глубокого обучения (ИНС) не стали популярными на этапе 1990-х годов:

1. Наши размеченные наборы данных были в тысячи раз меньше.

2. Наши компьютеры были в миллионы раз медленнее.
3. Мы глупо инициализировали веса сети.
4. Мы использовали неправильный тип функции активации нелинейности.

Все эти причины указывают на то, что обучающие сети с глубиной более двух скрытых слоев были бесполезной, если не вычислительной, невозможностью.

В текущем воплощении мы видим, что приливы изменились. Теперь у нас есть: 1. Более быстрые компьютеры. 2. Высокооптимизированное оборудование (например, графические процессоры).

3. Большие размеченные наборы данных порядка миллионов изображений.
4. Лучшее понимание функций инициализации весов и того, что работает/не работает.
5. Превосходные функции активации и понимание того, почему предыдущие функции нелинейности застопорились в исследованиях . Talk, Deep Learning, Self-Taught Learning и Unsupervised Feature Learning [38], теперь мы можем создавать более глубокие нейронные сети и обучать их на большем количестве данных.

По мере увеличения глубины сети растет и точность классификации. Такое поведение отличается от традиционных алгоритмов машинного обучения (то есть логистической регрессии, SVM, деревьев решений и т. д.), где мы достигаем плато производительности даже при увеличении доступных обучающих данных. Сюжет, вдохновленный докладом Эндрю Ng 2015 года «Что ученые, работающие с данными, должны знать о глубоком обучении» [39], можно увидеть на рис. 2.8, представляющем пример такого поведения.

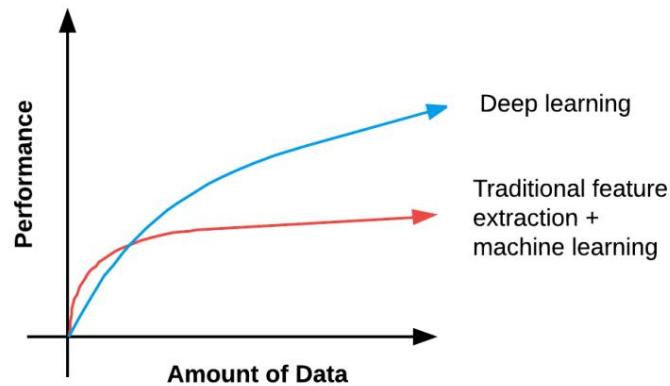


Рисунок 2.8: По мере увеличения объема данных, доступных для алгоритмов глубокого обучения, растет и точность , существенно превосходящая традиционные подходы к извлечению признаков и машинному обучению.

По мере увеличения объема обучающих данных алгоритмы нашей нейронной сети получают более высокую точность классификации, в то время как предыдущие методы в какой-то момент выходят на плато. Из-за связи между более высокой точностью и большим количеством данных мы также склонны связывать глубокое обучение с большими наборами данных .

При работе над вашими собственными приложениями глубокого обучения я предлагаю использовать следующее эмпирическое правило, чтобы определить, является ли ваша данная нейронная сеть глубокой:

1. Используете ли вы специализированную сетевую архитектуру, такую как сверточные нейронные сети, рекуррентные нейронные сети или сети с долговременной кратковременной памятью (LSTM)? Если да, то вы выполняете глубокое обучение.
2. Есть ли в вашей сети глубина > 2 ? Если да, вы занимаетесь глубоким обучением.
3. Есть ли в вашей сети глубина > 10 ? Если это так, вы выполняете очень глубокое обучение [40].

При всем этом старайтесь не увлекаться модными словечками, окружающими глубокое обучение и тем, что является/не является глубоким обучением. По сути, глубокое обучение прошло через ряд различных

воплощения за последние 60 лет, основанные на различных школах мысли, но каждая из этих школ мысли сосредоточена вокруг искусственных нейронных сетей, вдохновленных структурой и функцией мозга. Независимо от глубины сети, ширины или специализированной сетевой архитектуры вы по-прежнему выполняете машинное обучение с использованием искусственных нейронных сетей.

2.4 Резюме

В этой главе был рассмотрен сложный вопрос «Что такое глубокое обучение?».

Как мы выяснили, глубокое обучение существует с 1940-х годов, носит разные названия и воплощения, основанные на различных школах мысли и популярных направлениях исследований в данное время. По сути, глубокое обучение принадлежит к семейству искусственных нейронных сетей (ИНС), набору алгоритмов, которые изучают закономерности, вдохновленные структурой и функциями мозга.

Среди экспертов нет единого мнения о том, что именно делает нейронную сеть «глубокой»; однако мы знаем, что:

1. Алгоритмы глубокого обучения обучаются иерархически и, следовательно, накладываются на несколько слоев .
друг на друга, чтобы изучать все более абстрактные понятия.
2. Сеть должна иметь > 2 слоев, чтобы считаться «глубокой» (это мое анекдотическое мнение) .
на основе десятилетий исследований нейронных сетей).
3. Сеть с более чем 10 уровнями считается очень глубокой (хотя это число изменится, поскольку такие архитектуры, как ResNet, успешно обучаются с более чем 100 уровнями).

Если вы чувствуете себя немного сбитым с толку или даже подавленным после прочтения этой главы, не волнуйтесь — цель здесь состояла в том, чтобы просто дать очень общий обзор глубокого обучения и того, что именно означает «глубокое».

В этой главе также был представлен ряд понятий и терминов, с которыми вы, возможно, не знакомы, включая пиксели, края и углы — в нашей следующей главе будут рассмотрены основы этих типов изображений , и вы получите конкретную основу, на которую можно опереться. Затем мы начнем переходить к основам нейронных сетей, что позволит нам перейти к глубокому обучению и сверточным нейронным сетям позже в этой книге. Хотя эта глава, по общему признанию, была высокоуровневой, остальные главы этой книги будут чрезвычайно практическими, позволяя вам освоить глубокое изучение концепций компьютерного зрения.

3. Основы изображения

Прежде чем мы сможем начать создавать наши собственные классификаторы изображений, нам сначала нужно понять, что такое изображение . Мы начнем со строительных блоков изображения — пикселя.

Мы точно обсудим, что такое пиксель, как они используются для формирования изображения и как получить доступ к пикселям, представленным в виде массивов NumPy (как это делают почти все библиотеки обработки изображений в Python, включая OpenCV и scikit-image).

Глава завершится обсуждением соотношения сторон изображения и отношения, которое оно имеет при подготовке нашего набора данных изображения для обучения нейронной сети.

3.1 Пиксели: строительные блоки изображений

Пиксели — это необработанные строительные блоки изображения. Каждое изображение состоит из набора пикселей. Нет более тонкой детализации, чем пиксель.

Обычно пиксель считается «цветом» или «интенсивностью» света, который появляется в данном месте на нашем изображении. Если представить изображение как сетку, каждый квадрат содержит один пиксель. Например , взгляните на рисунок 3.1.

Изображение на рис. 3.1 выше имеет разрешение 1000×750 , что означает, что оно имеет ширину 1000 пикселей и высоту 750 пикселей. Мы можем концептуализировать изображение как (многомерную) матрицу. В этом случае наша матрица имеет 1000 столбцов (ширина) и 750 строк (высота). В целом, в нашем изображении $1000 \times 750 = 750\,000$ пикселей.

Большинство пикселей представлены двумя способами: 1. Оттенки серого/один канал 2. Цвет

В изображении в градациях серого каждый пиксель представляет собой скалярное значение от 0 до 255, где ноль соответствует «черному», а 255 — «белому». Значения от 0 до 255 представляют собой различные оттенки серого, где значения ближе к 0 темнее, а значения ближе к 255 светлее. Изображение с градиентом в градациях серого на рис. 3.2 демонстрирует более темные пиксели с левой стороны и все более светлые пиксели с правой стороны.

Цветные пиксели; однако обычно представлены в цветовом пространстве RGB (другие цветовые пространства не существуют, но выходят за рамки этой книги и не имеют отношения к глубокому обучению).

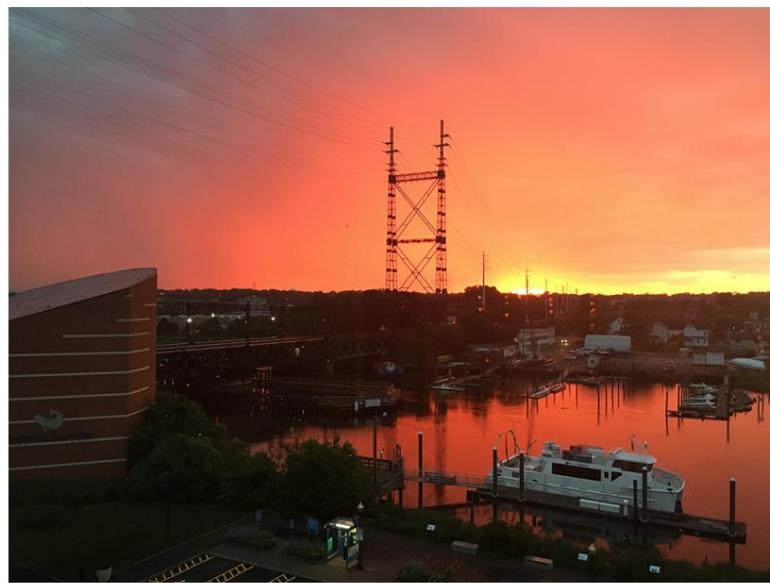


Рисунок 3.1: Это изображение имеет ширину 1000 пикселей и высоту 750 пикселей, всего $1000 \times 750 = 750\,000$ пикселей.



Рисунок 3.2: Градиент изображения, демонстрирующий изменение значений пикселей от черного (0) к белому (255).

 Если вы хотите узнать больше о цветовых пространствах (и основах компьютерного зрения и обработки изображений), см. [Практический курс Python и OpenCV вместе с курсом PyImageSearch Gurus](#).

Пиксели в цветовом пространстве RGB больше не являются скалярными значениями, как в изображении в градациях серого/одноканальном изображении — вместо этого пиксели представлены списком из трех значений: одно значение для красного компонента, одно для зеленого и еще одно для синего. Чтобы определить цвет в цветовой модели RGB, все, что нам нужно сделать, это определить количество красного, зеленого и синего, содержащихся в одном пикселе.

Каждый красный, зеленый и синий каналы могут иметь значения, определенные в диапазоне [0,255], всего 256 «оттенков», где 0 означает отсутствие представления, а 255 — полное представление. Учитывая, что значение пикселя должно быть только в диапазоне [0,255], мы обычно используем 8-битные целые числа без знака для представления интенсивности.

Как мы увидим, когда мы построим нашу первую нейронную сеть, мы часто будем предварительно обрабатывать наше изображение, выполняя вычитание среднего или масштабирование, что потребует от нас преобразования изображения в тип данных с плавающей запятой. Имейте это в виду, поскольку типы данных, используемые библиотеками, загружающими изображения с диска (например, OpenCV), часто должны быть преобразованы, прежде чем мы применим алгоритмы обучения непосредственно к изображениям.

Учитывая наши три значения Red, Green и Blue, мы можем объединить их в кортеж RGB в форме (красный, зеленый, синий). Этот кортеж представляет заданный цвет в цветовом пространстве RGB. Цветовое пространство RGB является примером аддитивного цветового пространства: чем больше каждого цвета добавляется, тем ярче становится пиксель и тем ближе он к белому. Мы можем визуализировать цветовое пространство RGB на рисунке 3.3 (слева). Как видите, добавление красного и зеленого приводит к желтому. Добавление красного и синего дает розовый. И

сложив вместе все три красного, зеленого и синего, мы получим белый цвет.

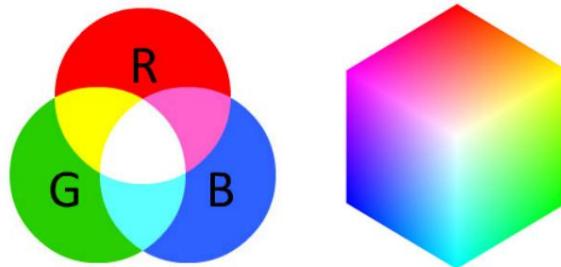


Рисунок 3.3: Слева: цветовое пространство RGB является аддитивным. Чем больше красного, зеленого и синего вы смешаете вместе, тем ближе получится белый. Справа: куб RGB.

Чтобы сделать этот пример более конкретным, давайте снова рассмотрим цвет «белый» — мы бы полностью заполнили каждое из красных, зеленых и синих ведер, вот так: (255, 255, 255). Затем, чтобы создать черный цвет, мы должны опустошить каждое из ведер (0, 0, 0), поскольку черный — это отсутствие цвета. Чтобы создать чистый красный цвет, мы должны полностью заполнить красное ведро (и только красное ведро): (255, 0, 0).

Цветовое пространство RGB также обычно визуализируется в виде куба (рис. 3.3, справа). Поскольку цвет RGB определяется как трехзначный кортеж, каждое значение которого находится в диапазоне [0,255], мы можем думать о кубе, содержащем $256 \times 256 \times 256 = 16\,777\,216$ возможных цветов, в зависимости от того, сколько красного, зеленого и синего помещено в каждое ведро.

В качестве примера давайте рассмотрим, сколько «красного», зеленого и синего нам потребуется для создания одного цвета (рис. 3.4, вверху). Здесь мы устанавливаем R=252, G=198, B=188, чтобы создать цветовой тон, похожий на кожу европеоида (возможно, полезно при создании приложения для определения количества кожи/плоти на изображении). Как мы видим, красный компонент сильно представлен с почти заполненным ведром. Зеленый и синий представлены почти поровну. Комбинируя эти цвета аддитивным образом, мы получаем цветовой тон, похожий на европеоидную кожу.

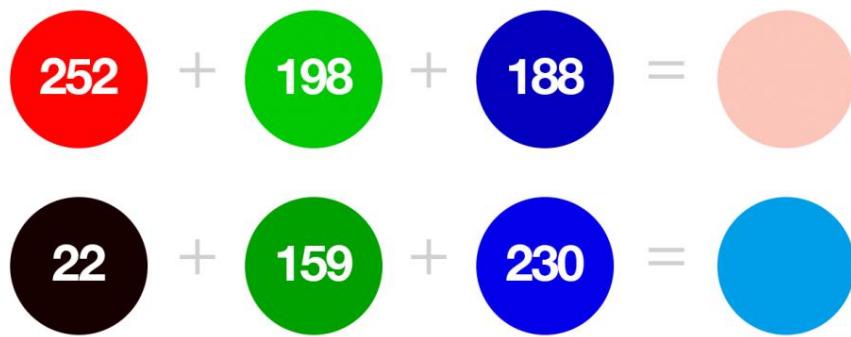


Рисунок 3.4: Вверху: пример объединения различных компонентов красного, зеленого и синего цветов для создания «кавказского телесного тона», возможно, полезного в программе обнаружения кожи. Внизу: создание определенного оттенка синего в логотипе «PyImageSearch» путем смешивания различных количеств красного, зеленого и синего.

Давайте попробуем другой пример, на этот раз установив R=22, G=159, B=230, чтобы получить оттенок синего, используемый в логотипе PyImageSearch (рис. 3.4, внизу). Здесь красный сильно недопредставлен с

значение 22 — ведро настолько пусто, что красное значение на самом деле кажется «черным». Зеленое ведро заполнено чуть более чем на 50%, в то время как синее ведро заполнено более чем на 90%, что явно является доминирующим цветом в представлении.

К основным недостаткам цветового пространства RGB относятся:

- Его аддитивная природа делает определение оттенков цвета немного неинтуитивным для людей.
- без использования инструмента «выбор цвета».
- Он не имитирует восприятие цвета людьми.

Несмотря на эти недостатки, почти все изображения, с которыми вы будете работать, будут представлены (по крайней мере, изначально) в цветовом пространстве RGB. Полный обзор цветовых моделей и цветовых пространств см . в курсе [PyImageSearch Gurus](#) [33].

3.1.1 Формирование изображения из каналов

Как мы теперь знаем, изображение RGB представлено тремя значениями, по одному для каждого из красного, зеленого и синего компонентов соответственно. Мы можем концептуализировать изображение RGB как состоящее из трех независимых матриц ширины W и высоты H , по одной для каждого компонента RGB, как показано на рис. 3.5. Мы можем объединить эти три матрицы, чтобы получить многомерный массив формы $W \times H \times D$, где D — глубина или количество каналов (для цветового пространства RGB $D = 3$):



Рисунок 3.5: Представление изображения в цветовом пространстве RGB, где каждый канал представляет собой независимую матрицу, которая при объединении формирует окончательное изображение.

Имейте в виду, что глубина изображения сильно отличается от глубины нейронной сети — этот момент станет понятен, когда мы начнем обучать наши собственные сверточные нейронные сети.

Однако на данный момент просто поймите, что подавляющее большинство изображений, с которыми вы будете работать:

- Представлены в цветовом пространстве RGB тремя каналами, каждый канал находится в диапазоне $[0,255]$. Заданный пиксель в изображении RGB представляет собой список из трех целых чисел: одно значение для красного, второе для зеленого и окончательное значение для синего.
- Программно определяется как трехмерный многомерный массив NumPy с шириной, высотой и глубиной.

3.2 Система координат изображения

Как упоминалось ранее в этой главе на рис. 3.1, изображение представляется в виде сетки пикселей. Чтобы сделать это более ясным, представьте нашу сетку в виде листа миллиметровой бумаги. Используя эту миллиметровую бумагу, исходная точка $(0,0)$ соответствует верхнему левому углу изображения. Когда мы движемся вниз и вправо, значения x и y увеличиваются.

На рис. 3.6 показано визуальное представление этого представления «в миллиметровке». Здесь у нас есть буква «I» на листе нашей миллиметровки. Мы видим, что это сетка 8×8 с общим размером 64 пикселя.

Важно отметить, что мы считаем с нуля, а не с единицы. Язык Python имеет нулевой индекс, а это означает, что мы всегда начинаем считать с нуля. Имейте это в виду, так как позже вы избежите путаницы (особенно если вы работаете в среде MATLAB).

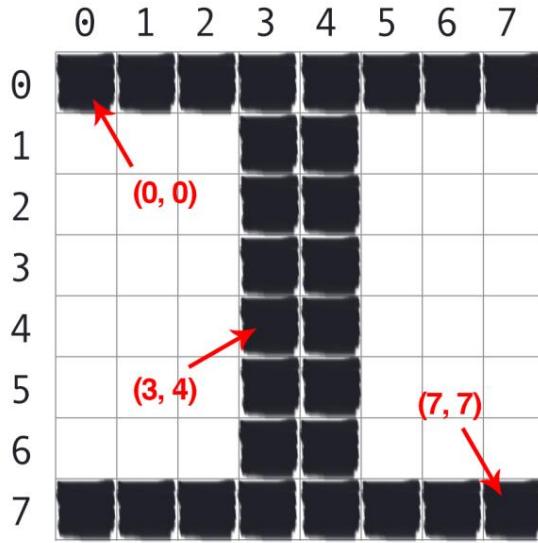


Рисунок 3.6: Буква «I» на листе миллиметровой бумаги. Доступ к пикселям осуществляется по их (x, y)-координатам, где мы идем на x столбцов вправо и на y строк вниз, имея в виду, что Python имеет нулевой индекс.

В качестве примера нулевой индексации рассмотрим 4 столбца пикселя справа и 5 строк вниз, индексированные точкой (3,4), опять же помня, что мы считаем с нуля, а не с единицы.

3.2.1 Изображения как массивы NumPy



Рисунок 3.7: Загрузка изображения с именем example.png с диска и отображение его на нашем экране с помощью OpenCV.

Библиотеки обработки изображений, такие как OpenCV и scikit-image, представляют изображения RGB в виде многомерных массивов NumPy с формой (высота, ширина, глубина). Читатели, которые впервые используют библиотеки обработки изображений, часто смущаются этим представлением — почему высота предшествует ширине, когда мы обычно сначала думаем об изображении с точки зрения ширины, а затем высоты?

Ответ связан с матричной записью.

При определении размеров матрицы мы всегда пишем ее как строки x столбцы. Количество строк в изображении — это его высота, а количество столбцов — это ширина изображения. Глубина все равно останется глубиной.

Поэтому, хотя представление .shape массива NumPy, представленного как (высота, ширина, глубина), может немного сбить с толку , это представление на самом деле имеет интуитивно понятный смысл при рассмотрении того, как матрица построена и аннотирована.

Например, давайте взглянем на библиотеку OpenCV и функцию cv2.imread , используемую для загрузки образ с диска и вывести его размеры:

```
1 import cv2 2
image = cv2.imread("example.png") 3
print(image.shape) 4 cv2.imshow("Image",
image) 5 cv2.waitKey(0)
```

Здесь мы загружаем изображение с именем example.png с диска и отображаем его на нашем экране, как скриншот с рис. 3.7 демонстрирует. Мой вывод терминала следующий:

```
$ Python load_display.py (248,
300, 3)
```

Это изображение имеет ширину 300 пикселей (количество столбцов), высоту 248 пикселей (количество строк) и глубину 3 (количество каналов). Чтобы получить доступ к отдельному значению пикселя из нашего изображения, мы используем простую индексацию массива NumPy:

```
1 (b, g, r) = image[20, 100] # доступ к пикселию с координатами x=100, y=20
2 (b, g, r) = image[75, 25] # доступ к пикселям с координатами x=25, y =75
3 (b, g, r) = image[90, 85] # доступ к пикселию с координатами x=85, y=90
```

Опять же, обратите внимание, как значение у передается перед значением x — этот синтаксис может показаться неудобным поначалу, но он согласуется с тем, как мы обращаемся к значениям в матрице: сначала мы указываем номер строки, а затем номер столбца. Оттуда нам дается кортеж, представляющий красный, зеленый и синий компоненты изображения.

3.2.2 Порядок RGB и BGR

Важно отметить,

что OpenCV хранит каналы RGB в обратном порядке. В то время как мы обычно думаем о красном, зеленом и синем, OpenCV фактически хранит значения пикселей в порядке синий, зеленый, красный.

Почему OpenCV делает это? Ответ просто исторические причины. Ранние разработчики библиотеки OpenCV выбрали цветовой формат BGR, потому что порядок BGR был популярен среди производителей камер и других разработчиков программного обеспечения в то время [41].

Проще говоря — этот заказ BGR был сделан по историческим причинам и выбору, с которым нам теперь приходится жить. Это небольшое предостережение, но его важно помнить при работе с OpenCV.

3.3 Масштабирование и соотношение сторон

Масштабирование или просто изменение размера — это процесс увеличения или уменьшения размера изображения по ширине и высоте. При изменении размера изображения важно помнить о соотношении сторон, которое



Рисунок 3.8: Слева: Исходное изображение. Вверх и вниз: результирующие искаженные изображения после изменения размера без сохранения соотношения сторон (т. е. отношения ширины к высоте изображения).

отношение ширины к высоте изображения. Игнорирование соотношения сторон может привести к тому, что изображения будут выглядеть сжатыми и искаженными, как на рис. 3.8.

Слева у нас исходное изображение. А сверху и снизу у нас есть два изображения, которые были искажены из-за не сохранения соотношения сторон. Конечным результатом является то, что эти изображения искажаются, хрустят и сжимаются. Чтобы предотвратить такое поведение, мы просто масштабируем ширину и высоту изображения на равные величины при изменении размера изображения.

С чисто эстетической точки зрения вы почти всегда хотите, чтобы соотношение сторон изображения сохранялось при изменении размера изображения, но это правило не всегда подходит для глубокого обучения. Большинство нейронных сетей и сверточных нейронных сетей, применяемых для классификации изображений, предполагают ввод фиксированного размера, а это означает, что размеры всех изображений, которые вы проходите через сеть, должны быть одинаковыми. Общие варианты размеров изображения по ширине и высоте, вводимых в сверточные нейронные сети, включают 32×32 , 64×64 , 224×224 , 227×227 , 256×256 и 299×299 .

Предположим, мы проектируем сеть, которая должна классифицировать изображения 224×224 ; однако наш набор данных состоит из изображений размером 312×234 , 800×600 и 770×300 , среди изображений других размеров — как мы должны предварительно обрабатывать эти изображения? Мы просто игнорируем соотношение сторон и занимаемся искажением (рис. 3.9, внизу слева)? Или мы разработаем другую схему изменения размера изображения, например, изменим размер изображения по его кратчайшему измерению, а затем возьмем центральную обрезку (рис. 3.9, внизу справа)?

Как мы видим в левом нижнем углу, соотношение сторон нашего изображения было проигнорировано, в результате чего изображение выглядит искаженным и «сморщенным». Затем в правом нижнем углу мы видим, что соотношение сторон изображения сохранено, но за счет обрезки части изображения. Это может быть особенно вредно для нашей системы классификации изображений, если мы случайно обрежем часть или весь объект, который хотим идентифицировать.

Какой метод лучше? Короче говоря, это зависит. Для некоторых наборов данных вы можете просто игнорировать соотношение сторон и сжимать, искажать и сжимать изображения перед их передачей по сети. Для других наборов данных целесообразно предварительно обработать их, изменив размер по кратчайшему измерению, а затем обрезав центр.

Мы рассмотрим оба этих метода (и способы их реализации) более подробно позже в этой книге, но важно представить эту тему сейчас, когда мы изучаем основы работы с изображениями.

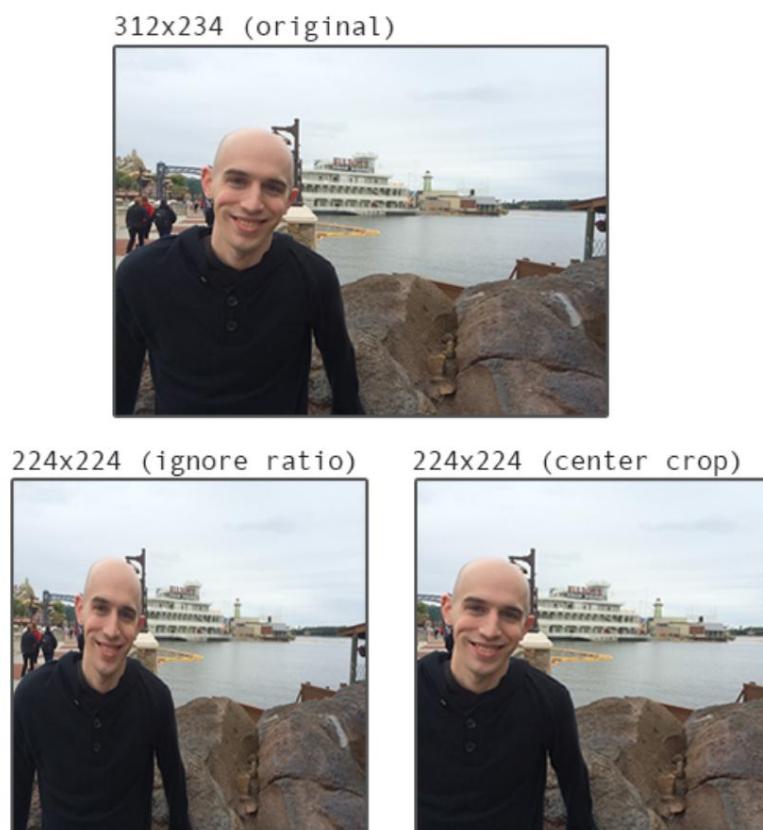


Рисунок 3.9: Вверху: исходное входное изображение. Внизу слева: изменение размера изображения до 224×224 пикселей без учета соотношения сторон. Внизу справа: изменение размера изображения 224 × 224 пикселей путем изменения размера сначала по кратчайшему измерению, а затем по центру кадрирования.

и как они представлены.

3.4 Резюме

В этой главе были рассмотрены основные строительные блоки изображения — пиксель. Мы узнали, что изображения в градациях серого/одноканальные представлены одним скаляром, интенсивностью/яркостью пикселя. Наиболее распространенным цветовым пространством является цветовое пространство RGB, в котором каждый пиксель изображения представлен тремя кортежами: по одному для каждого из красного, зеленого и синего компонентов соответственно.

Мы также узнали, что библиотеки компьютерного зрения и обработки изображений на языке программирования Python используют мощную библиотеку числовой обработки NumPy и, таким образом, представляют изображения в виде многомерных массивов NumPy. Эти массивы имеют форму (высота, ширина, глубина).

Сначала указывается высота, поскольку высота — это количество строк в матрице. Затем идет ширина, так как это количество столбцов в матрице. Наконец, глубина определяет количество каналов в изображении. В цветовом пространстве RGB глубина фиксируется на уровне `depth=3`.

Наконец, мы завершили эту главу, рассмотрев соотношение сторон изображения и роль, которую оно будет играть, когда мы изменим размер изображений в качестве входных данных для наших нейронных сетей и сверточных нейронных сетей. Для более подробного обзора цветовых пространств, системы координат изображения, изменения размера/соотношений сторон и других основ библиотеки OpenCV, пожалуйста, обратитесь к [Практическим Python и OpenCV \[8\]](#) и [курсу PyImageSearch Gurus \[33\]](#).

4. Основы классификации изображений

«Картинка стоит тысячи слов» — английская идиома.

Мы слышали эту поговорку бесчисленное количество раз в нашей жизни. Это просто означает, что сложную идею можно передать одним изображением. Независимо от того, изучаем ли мы линейный график нашего портфеля акций, смотрим на разворот предстоящего футбольного матча или просто изучаем искусство и мазки мастера живописи, мы постоянно воспринимаем визуальный контент, интерпретируем его значение и сохраняем знания для последующего использования.

Однако для компьютеров интерпретация содержимого изображения менее тривиальна — все, что видит наш компьютер, — это большая матрица чисел. Он понятия не имеет о мыслях, знаниях или значениях, которые изображение пытается передать.

Чтобы понять содержимое изображения, мы должны применить классификацию изображений, которая является задачей использования компьютерного зрения и алгоритмов машинного обучения для извлечения смысла из изображения. Это действие может быть таким же простым, как присвоение метки тому, что содержит изображение, или более сложным, как интерпретация содержимого изображения и возврат удобочитаемого предложения.

Классификация изображений — очень обширная область исследований, охватывающая широкий спектр методов. И с ростом популярности глубокого обучения она продолжает расти.

Настало время оседлать волну глубокого обучения и классификации изображений — те, кто успешно это сделает, будут щедро вознаграждены.

Классификация изображений и понимание изображений в настоящее время (и будут оставаться) самой популярной областью компьютерного зрения в течение следующих десяти лет. В будущем мы увидим, как такие компании, как Google, Microsoft, Baidu и другие, быстро приобретут успешные стартапы по анализу изображений. Мы будем видеть все больше и больше потребительских приложений на наших смартфонах, которые могут понимать и интерпретировать содержимое изображения. Даже войны, вероятно, будут вестись с использованием беспилотных летательных аппаратов, которые автоматически управляются с помощью алгоритмов компьютерного зрения.

В этой главе я представлю общий обзор того, что такое классификация изображений, а также множество проблем, которые должен преодолеть алгоритм классификации изображений. Мы также рассмотрим три различных типа обучения, связанных с классификацией изображений и машинным обучением.

Наконец, мы завершим эту главу, обсудив четыре этапа обучения сети глубокого обучения для классификации изображений и чем этот четырехэтапный конвейер сравнивается с традиционным,

созданный вручную конвейер извлечения признаков.

4.1 Что такое классификация изображений?

Классификация изображений, по своей сути, представляет собой задачу присвоения изображению метки из предопределенного набора категорий.

Практически это означает, что наша задача — проанализировать входное изображение и вернуть метку, которая классифицирует изображение. Метка всегда из предопределенного набора возможных категорий.

Например, предположим, что наш набор возможных категорий включает в себя:

Categories = {кошка, собака, панда}

Затем мы представляем следующее изображение (рис. 4.1) нашей системе классификации:



Рисунок 4.1: Цель системы классификации изображений — взять входное изображение и присвоить ему метку на основе предварительно определенного набора категорий.

Наша цель здесь — взять это входное изображение и присвоить ему метку из нашего набора категорий — в данном случае собака.

Наша система классификации также может присваивать изображению несколько меток с помощью вероятностей, например, собака: 95%; кошка: 4%; панда: 1%.

Более формально, учитывая наше входное изображение пикселей размером $W \times H$ с тремя каналами, красным, зеленым и синим соответственно, наша цель — взять изображение размером $W \times H \times 3 = N$ пикселей и выяснить, как правильно классифицировать содержимое Изображение.

4.1.1 Примечание по терминологии

При выполнении машинного обучения и глубокого обучения у нас есть набор данных, из которого мы пытаемся извлечь знания. Каждый пример/элемент в наборе данных (будь то данные изображения, текстовые данные, аудиоданные и т. д.) является точкой данных. Таким образом, набор данных представляет собой набор точек данных (рис. 4.2).

Наша цель — применить алгоритмы машинного обучения и глубокого обучения для обнаружения базовых закономерностей в наборе данных, что позволит нам правильно классифицировать точки данных, с которыми наш алгоритм еще не сталкивался. Потратите время, чтобы ознакомиться с этой терминологией: 1. В контексте

классификации изображений наш набор данных представляет собой набор изображений.

2. Таким образом, каждое изображение является точкой данных.

В оставшейся части книги я буду использовать термины «изображение» и «точка данных» как синонимы, поэтому имейте это в виду сейчас.

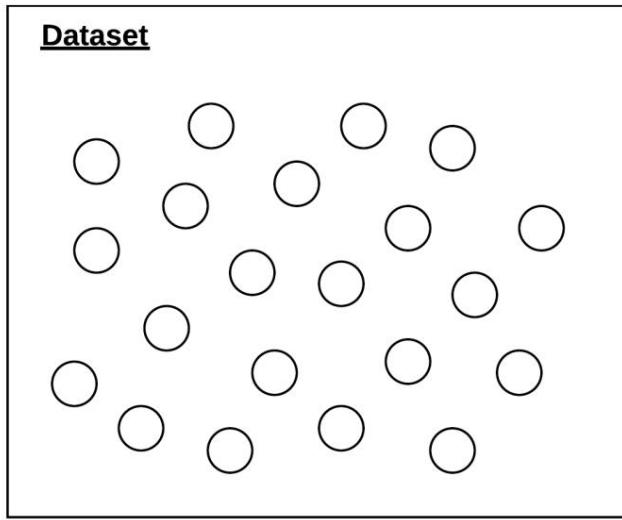


Рисунок 4.2: Набор данных (внешний прямоугольник) представляет собой набор точек данных (кружки).

4.1.2 Семантический разрыв

Взгляните на две фотографии (вверху) на рис. 4.3. Для нас должно быть довольно тривиально определить разницу между двумя фотографиями — ясно, что кошка слева и собака справа. Но все, что видит компьютер, — это две большие матрицы пикселей (внизу).

Учитывая, что все, что видит компьютер, — это большая матрица пикселей, мы приходим к проблеме семантического разрыва. Семантический разрыв — это разница между тем, как человек воспринимает содержимое изображения, и тем, как изображение может быть представлено так, чтобы компьютер мог понять процесс.

Опять же, быстрый визуальный осмотр двух фотографий выше может выявить разницу между двумя видами животных. Но на самом деле компьютер с самого начала понятия не имеет, что на изображении есть животные . Чтобы прояснить этот момент, взгляните на рис. 4.4, на котором изображен тихий пляж.

Мы можем описать изображение следующим образом:

- Пространственный: небо находится вверху изображения, а песок/океан внизу. • Цвет: небо темно-синее, вода в океане светлее неба, а песок загар.
- Текстура: Небо имеет относительно равномерный рисунок, а песок очень крупный.

Как нам закодировать всю эту информацию таким образом, чтобы ее мог понять компьютер? Ответ заключается в применении извлечения признаков для количественной оценки содержимого изображения. Извлечение признаков — это процесс получения входного изображения, применения алгоритма и получения вектора признаков (т. е. списка чисел), который количественно определяет наше изображение.

Чтобы выполнить этот процесс, мы можем рассмотреть возможность применения разработанных вручную функций, таких как HOG, LBP или других «традиционных» подходов к количественной оценке изображений. Другой метод, описанный в этой книге, заключается в применении глубокого обучения для автоматического изучения набора функций, которые можно использовать для количественной оценки и, в конечном счете, маркировки содержимого самого изображения.

Однако это не так просто... потому что, как только мы начинаем рассматривать изображения в реальном мире, мы сталкиваемся с множеством проблем.



151	121	1	93	165	204	14	214	28	235	29	142	142	75	22	109	111	28	6	5
62	67	17	234	27	221	37	189	141	137	168	41	206	100	70	219	127	114	191	
20	168	155	113	178	228	25	130	139	221	205	154	226	14	89	86	242	67	203	15
236	136	158	230	10	5	165	17	30	155	247	47	128	123	253	229	181	251	232	28
174	148	93	70	95	106	151	10	160	214	68	75	24	99	93	63	215	222	102	180
103	126	58	16	138	136	98	202	42	233	206	246	85	103	215	3	62	64	77	216
235	103	52	37	94	104	173	86	223	113	126	80	165	149	196	75	186	60	179	193
212	15	179	139	48	232	194	46	174	37	44	253	164	253	14	216	175	30	46	254
119	81	241	172	95	170	29	210	22	194	137	23	33	203	241	21	144	63	244	188
129	19	33	253	229	5	152	233	52	44	32	214	142	121	249	109	99	232	183	71
88	200	194	185	140	200	223	190	164	102	45	36	152	27	190	137	61	1	237	247
113	16	220	215	143	104	247	29	97	203	1	14	241	70	2	30	151	67	169	205
9	210	102	246	75	9	158	104	184	129	32	80	102	32	99	169	91	166	73	214
124	52	76	148	249	107	65	216	187	181	186	219	9	203	209	240	40	249	119	122
6	251	52	208	46	65	185	38	77	240	177	252	38	203	119	0	217	139	139	157
150	194	28	206	148	197	208	28	74	93	154	145	49	251	150	185	235	23	230	156
33	183	248	153	168	205	146	100	254	218	157	168	223	60	247	118	5	180	16	206
130	53	128	212	61	226	201	110	140	183	102	208	195	246	140	138	54	191	139	79
165	246	22	102	151	213	40	138	8	93	17	233	85	169	166	24	49	40	160	97
152	251	101	230	23	162	70	238	75	24	84	242	247	144	203	3	19	24	198	88
187	105	152	83	167	98	125	180	136	121	67	67	185	98	123	106	168	105	127	153
139	197	55	209	28	124	208	208	184	40	37	113	214	252	203	80	146	211	7	16
123	19	144	223	62	253	202	108	47	242	142	241	66	86	214	133	146	253	189	200
220	144	31	16	136	123	227	62	183	163	67	215	174	111	189	54	144	56	59	163

Рисунок 4.3: Вверху: наш мозг может четко видеть разницу между изображением, на котором изображена кошка, и изображением, на котором изображена собака. Внизу: Однако все, что «видит» компьютер, — это большая матрица чисел.

Разница между тем, как мы воспринимаем изображение, и тем, как это изображение представляется (матрицей чисел), называется семантическим разрывом.

4.1.3 Проблемы

Если семантического разрыва было недостаточно, мы также должны иметь дело с факторами вариации [10] в том, как выглядит изображение или объект. На рис. 4.5 представлена визуализация ряда этих факторов вариации.

Для начала у нас есть вариация точки обзора, когда объект можно ориентировать/вращать в нескольких измерениях в зависимости от того, как объект фотографируется и захватывается. Независимо от того, под каким углом мы снимаем этот Raspberry Pi, это все равно Raspberry Pi.

Мы также должны учитывать изменение масштаба . Вы когда-нибудь заказывали в Starbucks чашку кофе в стиле толл, гранде или венти? Технически это одно и то же — чашка кофе.

Но все они размером с чашку кофе. Кроме того, тот же кофе венти будет выглядеть совершенно иначе, если его сфотографировать вблизи, а не издалека . Наши методы классификации изображений должны быть терпимы к этим типам вариаций масштаба.

Одним из наиболее сложных для учета изменений является деформация. Для тех из вас, кто знаком с телесериалом «Гамби», мы можем видеть главного героя на изображении выше. Как следует из названия телешоу , этот персонаж эластичен, растягивается и способен искривлять свое тело в самых разных позах. Мы можем рассматривать эти изображения Гамби как тип деформации объекта — все изображения содержат характер Гамби; однако все они резко отличаются друг от друга.

Наша классификация изображений также должна быть в состоянии обрабатывать окклюзии, когда большие части изображения



Рисунок 4.4: При описании содержимого этого изображения мы можем сосредоточиться на словах, которые передают пространственную структуру, цвет и текстуру — то же самое верно для алгоритмов компьютерного зрения.

объекты, которые мы хотим классифицировать, скрыты от глаз на изображении (рис. 4.5). Слева у нас должна быть картинка с собакой. А справа у нас есть фотография той же собаки, но обратите внимание, как собака отдыхает под одеялом, скрытая от нашего взгляда. Собака по-прежнему отчетливо видна на обоих изображениях — просто на одном она более заметна, чем на другом. Алгоритмы классификации изображений по-прежнему должны обнаруживать и маркировать присутствие собаки на обоих изображениях.

Так же сложно, как упомянутые выше деформации и окклюзии, нам также необходимо обрабатывать изменения в освещении. Взгляните на кофейную чашку, снятую при стандартном и слабом освещении (рис. 4.5). Изображение слева было сфотографировано при стандартном верхнем освещении, а изображение справа — при очень слабом освещении. Мы по-прежнему изучаем ту же чашку, но, исходя из условий освещения, чашка выглядит совершенно по-другому (приятно, что вертикальный картонный шов чашки хорошо виден в условиях слабого освещения, но не при стандартном освещении).

Продолжая, мы также должны учитывать фоновый беспорядок. Вы когда-нибудь играли в игру «Где Уолдо?» (Или «Где Уолли?» для наших иностранных читателей.) Если да, то вы знаете, что цель игры — найти нашего любимого друга в красно-белой полосатой рубашке. Тем не менее, эти головоломки — больше, чем просто развлекательная детская игра — они также идеально отражают беспорядок на заднем плане. Эти изображения невероятно «шумны» и в них много всего происходит. Нас интересует только один конкретный объект на изображении; однако из-за всего этого «шума» нелегко отличить Уолдо/Уолли. Если нам это не просто сделать, то представьте, как тяжело компьютеру без смыслового понимания изображения!

Наконец, у нас есть внутриклассовая вариация. Каноническим примером внутриклассовой изменчивости компьютерного зрения является разнообразие стульев. От удобных стульев, которые мы используем, чтобы свернуться калачиком и почитать книгу, до стульев, стоящих вдоль нашего кухонного стола для семейных встреч, до ультрасовременных стульев в стиле ар-деко, которые можно найти в престижных домах, стул по-прежнему остается стулом, и наши алгоритмы классификации изображений должны уметь правильно классифицировать все эти варианты.

Вы начинаете чувствовать себя немного ошеломленным сложностью создания классификатора изображений? К сожалению, дальше становится только хуже — нашей системе классификации изображений недостаточно быть независимой устойчивой к этим вариациям, наша система также должна обрабатывать несколько вариаций, объединенных вместе!

Так как же объяснить такое невероятное количество вариаций объектов/изображений? В общем, постараемся сформулировать проблему как можно лучше. Мы делаем предположения относительно содержания наших изображений и к каким вариациям мы хотим быть терпимыми. Мы также учитываем масштабы нашей

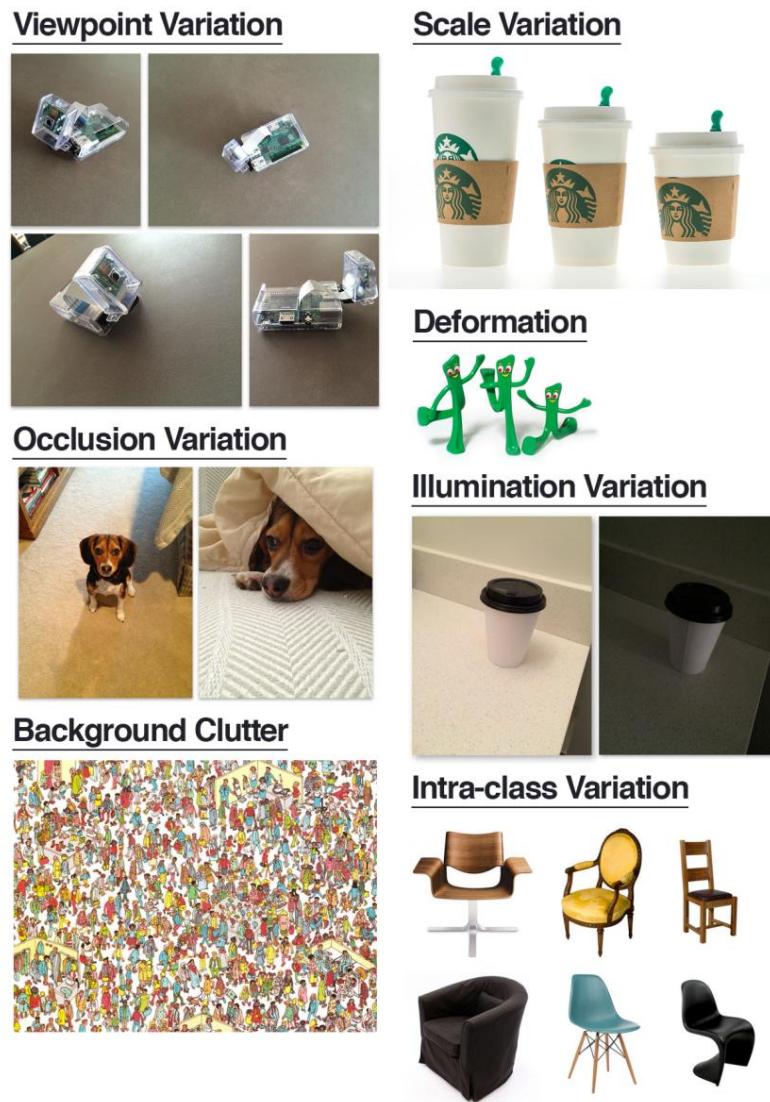


Рисунок 4.5: При разработке системы классификации изображений нам необходимо учитывать, как объект может выглядеть с разных точек обзора, условий освещения, окклюзии, масштаба и т. д.

проект – какова конечная цель? И что мы пытаемся построить?

Успешные системы компьютерного зрения, классификации изображений и глубокого обучения, развернутые в реальном мире, делают тщательные предположения и соображения еще до того, как будет написана хоть одна строка кода.

Если вы примите слишком широкий подход, например: «я хочу классифицировать и обнаружить каждый объект на моей кухне» (где могут быть сотни возможных объектов), то ваша система классификации вряд ли будет работать хорошо, если у вас нет многолетнего опыта. опыт создания классификаторов изображений — и даже в этом случае успех проекта не гарантируется.

Но если вы сформулируете свою проблему и сделаете ее узкой, например, «Я хочу распознавать только плиты и холодильники», то ваша система, скорее всего, будет точной и работоспособной, особенно если вы впервые работаете с классификацией изображений. и глубокое обучение.

Ключевым моментом здесь является то, что всегда следует учитывать область действия вашего классификатора изображений. Несмотря на то, что глубокое обучение и сверточные нейронные сети продемонстрировали значительную надежность и классификационную способность при решении различных задач, вы все равно должны сохранять масштаб своего проекта как можно дальше.

как можно плотнее и четче.

Имейте в виду, что ImageNet [42], де-факто стандартный эталонный набор данных для алгоритмов классификации изображений, состоит из 1000 объектов, с которыми мы сталкиваемся в повседневной жизни, и этот набор данных до сих пор активно используется исследователями, пытающимися повысить искусство для глубокого обучения вперед.

Глубокое обучение — это не магия. Наоборот, глубокое обучение похоже на пилю в вашем гараже — мощную и полезную при правильном использовании, но опасную, если использовать ее без должного внимания. На протяжении оставшейся части этой книги я буду направлять вас на вашем пути к глубокому обучению и указывать, когда вам следует использовать эти мощные инструменты, а когда вместо этого следует обратиться к более простому подходу (или упомянуть, если проблема не является разумной для изображения). классификация для решения).

4.2 Типы обучения

Существует три типа обучения, с которыми вы, вероятно, столкнетесь в своей карьере в машинном обучении и глубоком обучении: контролируемое обучение, неконтролируемое обучение и частично контролируемое обучение.

Эта книга посвящена обучению с учителем в контексте глубокого обучения. Тем не менее, описания всех трех типов обучения представлены ниже.

4.2.1 Контролируемое обучение

Представьте себе: вы только что закончили колледж со степенью бакалавра компьютерных наук. Вы молоды. Сломанный. И ищете работу в этой сфере — возможно, вы даже чувствуете себя потерянным в поисках работы.

Но прежде чем вы это узнаете, рекруттер Google найдет вас в LinkedIn и предложит вам вакансию. Работают над своим программным обеспечением Gmail. Вы собираетесь взять это? Более вероятный.

Несколько недель спустя вы подъезжаете к впечатляющему кампусу Google в Маунтин-Вью, штат Калифорния, пораженные захватывающими дух пейзажами, парком автомобилей Tesla на стоянке и почти нескончаемыми рядами изысканной еды в столовой.

Наконец вы садитесь за свой стол в широко открытом рабочем пространстве среди сотен других сотрудников . . . и тогда вы узнаете свою роль в компании. Вас наняли для создания программного обеспечения для автоматической классификации электронной почты как спама или не спама.

Как собираетесь достичь этой цели? Будет ли работать подход, основанный на правилах? Не могли бы вы написать серию операторов if/else , которые ищут определенные слова, а затем определяют, является ли электронное письмо спамом, на основе этих правил?

Вместо этого вам действительно нужно машинное обучение. Вам нужен обучающий набор, состоящий из самих писем вместе с их метками, в данном случае спам или не спам. Имея эти данные, вы можете анализировать текст (т. е. распределение слов) в электронном письме и использовать метки спам/не спам, чтобы научить классификатор машинного обучения, какие слова встречаются в спам-письме, а какие нет — и все это без необходимости чтобы вручную создать длинную и сложную серию операторов if/else.

Этот пример создания системы фильтрации спама является примером контролируемого обучения. Обучение с учителем, пожалуй, самый известный и изученный тип машинного обучения. Учитывая наши обучающие данные, модель (или «классификатор») создается в процессе обучения, в котором прогнозы делаются на входных данных, а затем исправляются, когда прогнозы неверны. Этот процесс обучения продолжается до тех пор, пока модель не достигнет некоторого желаемого критерия остановки, такого как низкий уровень ошибок или максимальное количество итераций обучения.

Общие алгоритмы обучения с учителем включают логистическую регрессию, машины опорных векторов. (SVM) [43, 44], случайные леса [45] и искусственные нейронные сети.

В контексте классификации изображений мы предполагаем, что наш набор данных изображений состоит из самих изображений вместе с соответствующей меткой класса, которую мы можем использовать для обучения нашему машинному обучению .

Этикетка	$R\mu$	$G\mu$	$B\mu$	$R\sigma$	$G\sigma$	$B\sigma$			
Кот	57,61	41,36	123,44	158,33	149,86	93,33	120,23	121,59	181,43
Кот	145,58	69,13	116,91	Кошка	124,15	193,35	65,77	23,63	193,702,70
Доба	100,289	162,827	107,81,19162	117	107,47147047	Доба	100,28	163,82	104,81
				149,49	197,41	18,99	87,15		

Таблица 4.1: Таблица данных, содержащая как метки класса (собака или кошка), так и векторы признаков для каждой точки данных (среднее значение и стандартное отклонение каждого канала красного, зеленого и синего цветов соответственно). Это пример контролируемой задачи классификации.

классификатор того, как «выглядит» каждая категория. Если наш классификатор делает неверный прогноз, мы можем применить методы, чтобы исправить его ошибку.

Различия между контролируемым, неконтролируемым и полуконтролируемым обучением лучше всего понять, рассмотрев пример в таблице 4.1. Первый столбец нашей таблицы — это метка, связанная с конкретным изображением. Остальные шесть столбцов соответствуют нашему вектору признаков для каждой точки данных — здесь мы решили количественно оценить содержимое нашего изображения, вычислив среднее значение и стандартное отклонение для каждого цветового канала RGB соответственно.

Наш алгоритм обучения с учителем будет делать прогнозы для каждого из этих векторов признаков, и если он делает неверный прогноз, мы попытаемся исправить его, сообщив ему, какая на самом деле правильная метка. Затем этот процесс будет продолжаться до тех пор, пока не будет достигнут желаемый критерий остановки, такой как точность, количество итераций процесса обучения или просто произвольное количество времени стены.

R

Чтобы объяснить различия между контролируемым, неконтролируемым и полуконтролируемым обучением, я решил использовать подход, основанный на признаках (т. е. среднее значение и стандартное отклонение цветовых каналов RGB) для количественной оценки содержания изображения. Когда мы начнем работать со свёрточными нейронными сетями, мы фактически пропустим этап извлечения признаков и будем использовать сами необработанные интенсивности пикселей. Поскольку изображения могут быть большими матрицами $M \times N$ (и, следовательно, не могут хорошо вписаться в этот пример электронной таблицы/таблицы), я использовал процесс извлечения признаков, чтобы помочь визуализировать различия между типами обучения.

4.2.2 Обучение без учителя В отличие

от обучения с учителем, обучение без учителя (иногда называемое самообучением) не имеет меток, связанных с входными данными, и поэтому мы не можем исправить нашу модель, если она делает неверный прогноз.

Возвращаясь к примеру с электронной таблицей, преобразовывая задачу обучения с учителем в задачу обучения без учителя так же просто, как удалить столбец «метка» (таблица 4.2).

Неконтролируемое обучение иногда считают «Святым Граалем» машинного обучения и классификации изображений. Когда мы рассматриваем количество изображений на Flickr или количество видео на YouTube, мы быстро понимаем, что в Интернете доступно огромное количество неразмеченных данных. Если бы мы могли заставить наш алгоритм изучать шаблоны из неразмеченных данных, нам не пришлось бы тратить много времени (и денег) на усердную маркировку изображений для контролируемых задач.

Большинство алгоритмов обучения без учителя наиболее успешны, когда мы можем изучить базовую структуру набора данных, а затем, в свою очередь, применить наши изученные функции к задаче обучения с учителем, где слишком мало помеченных данных, чтобы их можно было использовать.

Классические алгоритмы машинного обучения для обучения без учителя включают анализ основных компонентов (PCA) и кластеризацию k-средних. Конкретно для нейронных сетей мы видим автоэнкодеры, автокодировщики.

4.2 Типы обучения

R _μ	G _μ	B _μ	R _σ	G _σ	B _σ			
57,61	41,36	123,44	158,33	149,86	93,33			
120,23	121,59	181,43	145,58	69,13	116,91			
124,15	193,35	65,77	23,63	193,74	162,70			
100,28	163,82	104,81	19,62	117,07	21,11			
177,43	22,31	149,49	197,41	18,99	187,78			
149,73	87,17	187,97	50,27	87,15	36,65			

Таблица 4.2. Алгоритмы обучения без учителя пытаются изучить основные закономерности в наборе данных без меток классов. В этом примере мы удалили столбец метки класса, тем самым превратив эту задачу в проблему неконтролируемого обучения .

Этикетка	R _μ	G _μ	B _μ	R _σ	G _σ	B _σ		
Кошка	57,61	41,36	123,44	158,33	149,86	93,33		
?	120,23	121,59	181,43	145,58	69,13	116,91		
?	124,15	193,35	65,77	23,63	193,74	162,70		
Собака	100,28	163,82	104,81	19,62	117,07	21,11		
?	187,78	177,43	22,31		149,49	197,41	18,99	87,15
Собака	149,73	87,17	187,97	50,27	87,15	36,65		

Таблица 4.3: При полуkontролируемом обучении у нас есть только метки для подмножества изображения/векторы объектов и должны попытаться пометить другие точки данных, чтобы использовать их в качестве дополнительного обучения. данные.

Организационные карты (SOM) и теория адаптивного резонанса применяются к обучению без учителя.

Неконтролируемое обучение — чрезвычайно активная область исследований, которую еще предстоит решить. Мы не сосредотачиваемся на неконтролируемом обучении в этой книге.

4.2.3 Полуконтролируемое обучение

Итак, что произойдет, если у нас есть только некоторые метки, связанные с нашими данными, и нет меток для Другие? Можем ли мы применить некий гибрид контролируемого и неконтролируемого обучения и при этом быть в состоянии классифицировать каждую из точек данных? Оказывается, да – нужно только подать заявку полуkontролируемое обучение.

Возвращаясь к нашему примеру с электронной таблицей, предположим, что у нас есть метки только для небольшой части нашего исходные данные (табл. 4.3). Наш алгоритм обучения с получателем будет использовать известные фрагменты данных, проанализируйте их и попытайтесь пометить каждую из непомеченных точек данных для использования в качестве дополнительных обучающих данных. Этот процесс может повторяться в течение многих итераций по мере того, как полууправляемый алгоритм изучает «структурку» данных, чтобы делать более точные прогнозы и генерировать более надежные обучающие данные.

Полууправляемое обучение особенно полезно в компьютерном зерении, где часто требуется много времени, утомительно и дорого (по крайней мере, с точки зрения человека-часов) маркировать каждый отдельный элемент. изображение в нашем тренировочном наборе. В случаях, когда у нас просто нет времени или ресурсов для маркировки каждого отдельное изображение, мы можем пометить лишь небольшую часть наших данных и использовать обучение с получателем маркировать и классифицировать остальные изображения.

Алгоритмы полууправляемого обучения часто обменивают меньшие размеченные наборы входных данных на некоторое допустимое снижение точности классификации. Как правило, более точно обозначенное обучение под наблюдением алгоритм обучения, тем более точные прогнозы он может делать (особенно это касается глубоких алгоритмы обучения).

По мере уменьшения количества обучающих данных точность неизбежно страдает. Полуконтролируемое обучение

принимает во внимание эту взаимосвязь между точностью и объемом данных и пытается удерживать точность классификации в допустимых пределах, при этом резко сокращая объем обучающих данных, необходимых для построения модели. Конечным результатом является точный классификатор (но обычно не такой точный, как контролируемый классификатор) с меньшими усилиями и данными для обучения. Популярные варианты полуkontrolируемого обучения включают распространение меток [46], распространение меток [47], лестничные сети [48] и совместное обучение/совместное обучение [49].

Опять же, в этой книге мы в первую очередь сосредоточимся на обучении с учителем, поскольку как обучение без учителя, так и полууправляемое обучение в контексте глубокого обучения для компьютерного зрения все еще являются очень активными темами исследований без четких указаний о том, какие методы использовать.

4.3 Конвейер классификации глубокого обучения

Основываясь на наших предыдущих двух разделах о классификации изображений и типах алгоритмов обучения, вы, возможно, начинаете чувствовать себя немного сбитыми с толку новыми терминами, соображениями и тем, что выглядит непреодолимым количеством вариаций в построении классификатора изображений, но правда в том, что создание классификатора изображений довольно просто, если вы понимаете процесс.

В этом разделе мы рассмотрим важный сдвиг в мышлении, который необходимо предпринять при работе с машинным обучением. Далее я рассмотрю четыре этапа создания классификатора изображений на основе глубокого обучения, а также сравню и сопоставлю традиционное машинное обучение на основе признаков со сквозным глубоким обучением.

4.3.1 Изменение мышления

Прежде чем мы углубимся во что-то сложное, давайте начнем с того, с чем мы все (скорее всего) знакомы: с последовательностью Фибоначчи.

Последовательность Фибоначчи — это ряд чисел, где следующее число последовательности находится путем суммирования двух целых чисел перед ним. Например, при заданной последовательности 0, 1, 1 следующее число находится путем сложения $1 + 1 = 2$. Аналогично, при заданных 0, 1, 1, 2 следующее целое число в последовательности равно $1 + 2 = 3$.

Следуя этому шаблону, первая горстка чисел в последовательности выглядит следующим образом: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...

Конечно, мы также можем определить этот шаблон в (крайне неоптимизированной) функции Python, используя рекурсию:

```
1 >>> по определению фиб(n):
2 ...     если n == 0:
3 ...         вернуть 0
4 ...     Элиф n == 1:
5 ...         вернуть 1
6 ...     еще:
7 ...         вернуть фиб (n-1) + фиб (n-2)
8 ...
9 >>>
```

Используя этот код, мы можем вычислить n -е число в последовательности, предоставив значение n для функция Фибоначчи. Например, давайте вычислим 7-е число в последовательности Фибоначчи:

```
9 >>> выдумка(7)
10 13
```

И 13-й номер:

4.3 Конвейер классификации глубокого обучения

11 >>> выдумка(13)

12 [233](#)*И, наконец, 35-е число:*

13 >>> выдумка(35)

14 [9227465](#)

Как видите, последовательность Фибоначчи проста и является примером семейства функций, которые: 1. Принимают ввод, возвращают вывод.

2. Процесс четко определен.
3. Правильность вывода легко проверить.
4. Хорошо подходит для покрытия кода и наборов тестов.

В общем, вы, вероятно, написали тысячи и тысячи подобных процедурных функций в своей жизни. Независимо от того, вычисляете ли вы последовательность Фибоначчи, извлекаете данные из базы данных или вычисляете среднее значение и стандартное отклонение из списка чисел, все эти функции четко определены и легко проверяются на правильность.

К сожалению, это не относится к глубокому обучению и классификации изображений!

Вспомним из раздела 4.1.2, где мы рассматривали изображения кошки и собаки, воспроизведенные для удобства на рис. 4.6. Теперь представьте, что вы пытаетесь написать процедурную функцию, которая может не только отличить эти две фотографии, но и любую фотографию кошки и собаки. Как бы вы подошли к выполнению этой задачи? Вы бы проверили значения отдельных пикселей в различных (x, y)-координатах?

Написать сотни операторов if/else ? И как бы вы поддерживали и проверяли правильность такой массивной системы, основанной на правилах? Короткий ответ: вы этого не сделаете.



Рисунок 4.6: Как вы могли бы написать программу, которая распознает разницу между собаками и кошками на изображениях? Не могли бы вы проверить значения отдельных пикселей? Принять подход, основанный на правилах? Пытаться написать (и поддерживать) сотни операторов if/else?

В отличие от написания алгоритма для вычисления последовательности Фибоначчи или сортировки списка чисел, не интуитивно и не очевидно, как создать алгоритм, чтобы отличить изображения кошек от собак. Таким образом, вместо того, чтобы пытаться построить основанную на правилах систему для описания того, как «выглядит» каждая категория, мы можем вместо этого использовать подход, основанный на данных, путем предоставления примеров того, как выглядит каждая категория, а затем научить наш алгоритм распознавать разницу между категориями, используя эти примеры.

Мы называем эти примеры нашим обучающим набором данных помеченных изображений, где каждая точка данных в нашем обучающем наборе данных состоит из: 1. Изображение

2. Ярлык/категория (например, собака, кошка, панда и т. д.) изображения

Опять же, важно, чтобы каждое из этих изображений имело ярлыки, связанные с ними, потому что наш контролируемый алгоритм обучения должен будет видеть эти ярлыки, чтобы «обучить себя». как распознать каждую категорию. Помня об этом, давайте продолжим работу над четырьмя этапами построения модели глубокого обучения.

4.3.2 Шаг № 1. Соберите свой набор данных Первым

компонентом построения сети глубокого обучения является сбор исходного набора данных. Нам нужны сами изображения, а также метки, связанные с каждым изображением. Эти ярлыки должны исходить из конечного набора категорий, например: категории = собака, кошка, панда.

Кроме того, количество изображений для каждой категории должно быть примерно одинаковым (т. е. одинаковое количество примеров в каждой категории). Если у нас будет в два раза больше изображений кошек, чем изображений собак, и в пять раз больше изображений панд, чем изображений кошек, тогда наш классификатор естественным образом будет склоняться к чрезмерной подгонке под эти широко представленные категории.

Дисбаланс классов — распространенная проблема в машинном обучении, и существует несколько способов ее преодоления. Мы обсудим некоторые из этих методов позже в этой книге, но имейте в виду, что лучший способ избежать проблем с обучением из-за дисбаланса классов — это просто полностью избежать дисбаланса классов.

4.3.3 Шаг № 2: Разделите набор данных

Теперь, когда у нас есть исходный набор данных, нам нужно разделить его на две части:

1. Обучающий набор
2. Тестовый набор

Обучающий набор используется нашим классификатором, чтобы «узнать», как выглядят каждая категория, делая прогнозы на входных данных, а затем исправляя себя, когда прогнозы неверны. После обучения классификатора мы можем оценить его работу на тестовом наборе.

Чрезвычайно важно, чтобы обучающая и тестовая выборки были независимы друг от друга и не пересекались! Если вы используете свой тестовый набор как часть своих обучающих данных, то ваш классификатор имеет несправедливое преимущество, поскольку он уже видел тестовые примеры раньше и «извлек уроки» из них. Вместо этого вы должны полностью отделить этот тестовый набор от процесса обучения и использовать его только для оценки вашей сети.

Общие размеры разделения для обучающих и тестовых наборов включают 66,6% 33,3%, 75%/25% и 90%/10%. соответственно. (Рисунок 4.7):

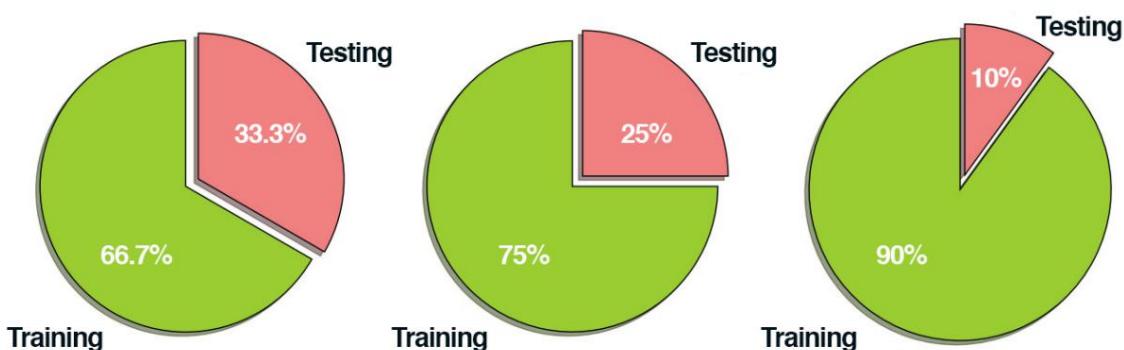


Рисунок 4.7: Примеры общего разделения данных обучения и тестирования.

Такое разбиение данных имеет смысл, но что, если у вас есть параметры для настройки? Нейронные сети имеют ряд ручек и рычагов (например, скорость обучения, затухание, регуляризация и т. д.), которые необходимо настроить.

и набирается для получения оптимальной производительности. Мы будем называть эти типы параметров гиперпараметрами, и очень важно, чтобы они были установлены правильно.

На практике нам нужно протестировать множество этих гиперпараметров и определить набор параметров, который работает лучше всего. У вас может возникнуть соблазн использовать данные тестирования для настройки этих значений, но опять же, это главное нет-нет! Набор тестов используется только для оценки производительности вашей сети.

Вместо этого вы должны создать третье разделение данных, называемое проверочным набором. Этот набор данных (обычно) поступает из обучающих данных и используется как «фальшивые тестовые данные», чтобы мы могли настроить наши гиперпараметры. Только после того, как мы определили значения гиперпараметров с помощью проверочного набора, мы переходим к сбору окончательных результатов точности в данных тестирования.

Обычно мы выделяем примерно 10-20% обучающих данных для проверки. Если разделение ваших данных на фрагменты кажется сложным, на самом деле это не так. Как мы увидим в следующей главе, это довольно просто и может быть реализовано всего одной строкой кода благодаря библиотеке scikit-learn.

4.3.4 Шаг № 3: Обучите свою сеть

Учитывая наш тренировочный набор изображений, теперь мы можем обучить нашу сеть. Цель здесь состоит в том, чтобы наша сеть научилась распознавать каждую из категорий в наших размеченных данных. Когда модель совершает ошибку, она учится на этой ошибке и совершенствуется.

Итак, как же происходит настоящее «обучение»? В общем, мы применяем форму градиентного спуска, как обсуждалось в главе 9. Оставшаяся часть этой книги посвящена демонстрации того, как обучать нейронные сети с нуля, поэтому мы отложим подробное обсуждение процесса обучения до тех пор.

4.3.5 Шаг № 4: Оценка

Наконец, нам нужно оценить нашу обученную сеть. Для каждого из изображений в нашем тестовом наборе мы представляем их сети и просим ее предсказать, что, по ее мнению, является меткой изображения. Затем мы сводим в таблицу прогнозы модели для изображения в тестовом наборе.

Наконец, эти прогнозы модели сравниваются с достоверными метками из нашего набора тестов. Метки достоверности представляют собой то, что на самом деле представляет собой категория изображения. Оттуда мы можем вычислить количество прогнозов, которые наш классификатор получил правильно, и вычислить совокупные отчеты, такие как точность, полнота и f-мера, которые используются для количественной оценки производительности нашей сети в целом.

4.3.6 Обучение на основе признаков по сравнению с глубоким обучением для классификации изображений

В традиционном подходе к классификации изображений, основанном на признаках, между шагами № 2 и № 3 фактически вставлен шаг — этот шаг представляет собой извлечение признаков. На этом этапе мы применяем разработанные вручную алгоритмы, такие как HOG [32], LBP [21] и т. д., для количественной оценки содержимого изображения на основе определенного компонента изображения, которое мы хотим закодировать (т. е. формы, цвета, текстуры и т. д.). Учитывая эти особенности, мы затем приступаем к обучению нашего классификатора и его оценке.

При построении сверточных нейронных сетей мы можем пропустить шаг извлечения признаков. Причина этого в том, что CNN являются сквозными моделями. Мы представляем необработанные входные данные (пиксели) в сеть. Затем сеть изучает фильтры внутри своих скрытых слоев, которые можно использовать для различения классов объектов. Результатом работы сети является распределение вероятностей по меткам классов.

Один из захватывающих аспектов использования CNN заключается в том, что нам больше не нужно возиться с ручными инженерными функциями — вместо этого мы можем позволить нашей сети изучить эти функции. Однако этот компромисс имеет свою цену. Обучение CNN может быть нетривиальным процессом, поэтому будьте готовы потратить значительное время на ознакомление с опытом и проведение множества экспериментов, чтобы определить, что работает, а что нет.

4.3.7 Что происходит, когда мои прогнозы неверны?

Неизбежно вы будете обучать сеть глубокого обучения на своем обучающем наборе, оценивать ее на своем тестовом наборе (обнаружив, что она обеспечивает высокую точность), а затем применять ее к изображениям, которые находятся за пределами как вашего обучающего, так и тестового набора — только для того, чтобы обнаружить, что сеть работает плохо.

Эта проблема называется обобщением, способностью сети обобщать и правильно предсказать метку класса изображения, которое не существует, как часть его обучающих или тестовых данных.

Способность сети к обобщению — буквально самый важный аспект исследований глубокого обучения — если мы сможем обучать сети, которые могут обобщать внешние наборы данных без повторного обучения или тонкой настройки, мы добьемся больших успехов в машинном обучении, позволяя сетям для повторного использования в различных областях. Способность сети к обобщению будет много раз обсуждаться в этой книге, но я хотел поднять эту тему сейчас, поскольку вы неизбежно столкнетесь с проблемами обобщения, особенно по мере того, как будете изучать основы глубокого обучения.

Вместо того, чтобы расстраиваться из-за того, что ваша модель неправильно классифицирует изображение, рассмотрите набор факторов вариации, упомянутых выше. Точно ли ваш обучающий набор данных отражает примеры этих факторов вариации? Если нет, вам нужно будет собрать больше обучающих данных (и прочитать оставшуюся часть этой книги, чтобы узнать о других методах борьбы с обобщениями).

4.4 Резюме

В этой главе мы узнали, что такое классификация изображений и почему это такая сложная задача для компьютеров (даже несмотря на то, что люди делают это интуитивно без каких-либо усилий). Затем мы обсудили три основных типа машинного обучения: обучение с учителем, обучение без учителя, обучение с частичным учителем — в этой книге основное внимание уделяется обучению с учителем, где у нас есть как обучающие примеры, так и связанные с ними метки классов. Полууправляемое обучение и неконтролируемое обучение являются открытыми областями исследований в области глубокого обучения (и машинного обучения в целом).

Наконец, мы рассмотрели четыре этапа конвейера классификации глубокого обучения. Эти шаги включают сбор вашего набора данных, разделение ваших данных на этапы обучения, тестирования и проверки, обучение вашей сети и, наконец, оценку вашей модели.

В отличие от традиционных подходов, основанных на признаках, которые требуют от нас использования созданных вручную алгоритмов для извлечения признаков из изображения, модели классификации изображений, такие как сверточные нейронные сети, представляют собой сквозные классификаторы, которые внутренне изучают признаки, которые можно использовать для различия между ними. классы изображений.

5. Наборы данных для классификации изображений

На данный момент мы привыкли к основам конвейера классификации изображений, но прежде чем мы углубимся в какой-либо код, рассматривающий, как на самом деле взять набор данных и построить классификатор изображений, давайте сначала рассмотрим наборы данных, которые вы увидите в Deep Learning для Компьютерное зрение с Python.

Некоторые из этих наборов данных по существу «решены», что позволяет нам получать классификаторы с чрезвычайно высокой точностью (точность > 95%) с небольшими усилиями. Другие наборы данных представляют категории компьютерного зрения, а проблемы глубокого обучения до сих пор остаются открытыми темами для исследований и далеки от решения. Наконец, некоторые наборы данных являются частью соревнований и задач по классификации изображений (например, Kaggle Dogs vs. Cats и cs231n Tiny ImageNet 200).

Важно просмотреть эти наборы данных сейчас, чтобы у нас было общее представление о проблемах, с которыми мы можем столкнуться при работе с ними в последующих главах.

5.1 MNIST



Рисунок 5.1: Пример набора данных MNIST. Цель этого набора данных — правильно классифицировать рукописные цифры от 0 до 9.

Набор данных MNIST («NIST» означает Национальный институт стандартов и технологий, а «M» означает «модифицированный», поскольку данные были предварительно обработаны, чтобы уменьшить нагрузку на обработку компьютерного зрения и сосредоточиться исключительно на задаче распознавания цифр) один из наиболее хорошо изученных наборов данных в литературе по компьютерному зрению и машинному обучению.

Цель этого набора данных — правильно классифицировать рукописные цифры от 0 до 9. Во многих случаях этот набор данных является эталоном, стандартом, по которому оцениваются алгоритмы машинного обучения. Фактически,

MNIST настолько хорошо изучен, что Джекфри Хинтон описал набор данных как «дрозофилу машинного обучения» [10] (дрозофилы — род плодовых мушек), сравнивая, как начинающие исследователи-биологи используют этих плодовых мушек, поскольку их легко культивировать в массовом порядке, имеют короткое время генерации и легко получают мутации.

В том же духе набор данных MNIST представляет собой простой набор данных для ранних практиков глубокого обучения, чтобы получить «первый вкус» обучения нейронной сети без особых усилий (очень легко получить > 97% точности классификации) — обучение модели нейронной сети на MNIST очень похож на «Hello, World» в машинном обучении.

Сам MNIST состоит из 60 000 обучающих изображений и 10 000 тестовых изображений. Каждый вектор признаков имеет размерность 784, что соответствует интенсивности пикселей изображения в градациях серого 28×28 . Эти интенсивности пикселей в градациях серого представляют собой целые числа без знака, попадающие в диапазон [0,255]. Все цифры размещены на черном фоне, а передний план — белый и оттенки серого. Учитывая эти необработанные интенсивности пикселей, наша цель — научить нейронную сеть правильно классифицировать цифры.

В первую очередь мы будем использовать этот набор данных в первых главах Starter Bundle, чтобы помочь нам «намочить ноги» и изучить основы нейронных сетей.

5.2 Животные: собаки, кошки и панды



Рисунок 5.2: Выборка набора данных о трех классах животных, состоящего из 1000 изображений для каждого класса собак, кошек и панд соответственно, всего 3000 изображений.

Цель этого набора данных — правильно классифицировать изображение как содержащее собаку, кошку или панду. Набор данных Animals, содержащий всего 3000 изображений, должен стать еще одним «вводным» набором данных, который мы можем быстро обучить модели глубокого обучения на нашем ЦП или ГП и получить разумную точность.

В главе 10 мы будем использовать этот набор данных, чтобы продемонстрировать, как использование пикселей изображения в качестве вектора признаков не преобразуется в высококачественную модель машинного обучения, если мы не используем использование сверточной нейронной сети (CNN).

Изображения для этого набора данных были собраны путем выборки изображений Kaggle Dogs vs. Cats вместе с набором данных ImageNet для примеров панд. Набор данных Animals в основном используется только в Starter Bundle.

5.3 СИФАР-10

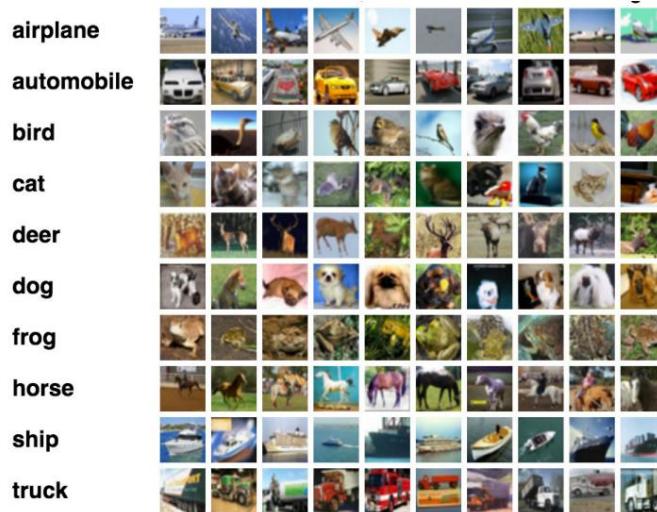


Рисунок 5.3: Примеры изображений из набора данных CIFAR-10 для десяти классов.

Как и MNIST, CIFAR-10 считается еще одним стандартным эталонным набором данных для классификации изображений в литературе по компьютерному зрению и машинному обучению. CIFAR-10 состоит из 60 000 изображений $32 \times 32 \times 3$ (RGB), что дает размерность вектора признаков 3072.

Как следует из названия, CIFAR-10 состоит из 10 классов, в том числе: самолеты, автомобили, птицы, кошки, олени, собаки, лягушки, лошади, корабли и грузовики.

В то время как довольно легко обучить модель, которая обеспечивает точность классификации $> 97\%$ в MNIST, значительно сложнее получить такую модель для CIFAR-10 (и его старшего брата, CIFAR-100) [50].

Проблема возникает из-за резкого различия в том, как выглядят объекты. Например, мы больше не можем предполагать, что изображение, содержащее зеленый пиксель с заданной координатой (x, y) , является лягушкой. Этот пиксель может быть частью фона леса, в котором есть олень. Или пиксель может быть просто цветом зеленого грузовика.

Эти предположения резко контрастируют с набором данных MNIST, где сеть может изучать предположения относительно пространственного распределения интенсивности пикселей. Например, пространственное распределение пикселей переднего плана с номером 1 существенно отличается от 0 или 5.

Несмотря на небольшой набор данных, CIFAR-10 по-прежнему регулярно используется для сравнительного анализа новой архитектуры CNN. Мы будем использовать CIFAR-10 как в наборе Starter Bundle, так и в наборе Practitioner Bundle.

5.4 УЛЫБКИ

Как следует из названия, набор данных SMILES [51] состоит из изображений лиц, которые либо улыбаются, либо не улыбаются. Всего в наборе данных 13 165 изображений в градациях серого, каждое из которых имеет размер 64×64 .

Изображения в этом наборе данных плотно обрезаны вокруг лица, что позволяет нам разрабатывать алгоритмы машинного обучения, которые сосредоточены исключительно на задаче распознавания улыбки. Отделение предварительной обработки компьютерного зрения от машинного обучения (особенно для эталонных наборов данных) является общей тенденцией, которую вы будете

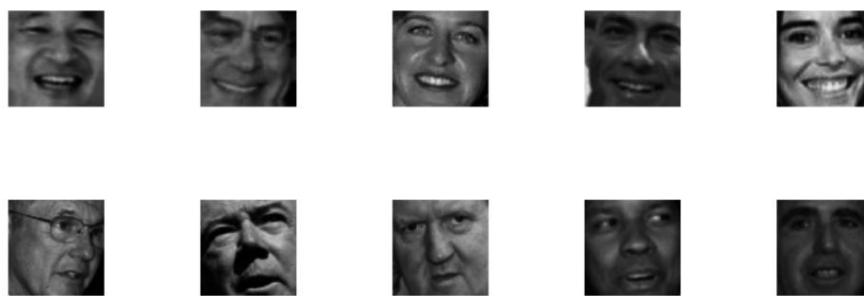


Рисунок 5.4: Вверху: Примеры «улыбающихся» лиц. Внизу: образцы «не улыбающихся» лиц. Позже мы обучим сверточную нейронную сеть распознавать улыбающиеся и не улыбающиеся лица в видеопотоках в реальном времени.

см. при обзоре популярных эталонных наборов данных. В некоторых случаях несправедливо предполагать, что исследователь машинного обучения достаточно знаком с компьютерным зрением, чтобы должным образом предварительно обработать набор данных изображений до применения собственных алгоритмов машинного обучения.

Тем не менее, эта тенденция быстро меняется, и предполагается, что любой практик, заинтересованный в применении машинного обучения к задачам компьютерного зрения, имеет хотя бы элементарные знания в области компьютерного зрения. Эта тенденция сохранится и в будущем, поэтому, если вы планируете изучать глубокое обучение для компьютерного зрения в любой степени, обязательно дополните свое образование небольшим количеством компьютерного зрения, даже если это только основы.

Если вы обнаружите, что вам нужно улучшить свои навыки компьютерного зрения, взгляните на [Практический Python и OpenCV](#) [8].

5.5 Kaggle: собаки против кошек

Задача «Собаки против кошек» является частью конкурса Kaggle по разработке алгоритма обучения, который может правильно классифицировать изображение как содержащее собаку или кошку. В общей сложности 25 000 изображений предоставляются для обучения вашего алгоритма с различными разрешениями изображений. Образец набора данных можно увидеть на рис . 5.5.

То, как вы решите предварительно обработать свои изображения, может привести к различным уровням производительности, снова демонстрируя, что опыт работы в области компьютерного зрения и основ обработки изображений будет иметь большое значение при изучении глубокого обучения.

Мы будем использовать этот набор данных в пакете Practitioner Bundle, когда я буду демонстрировать, как претендовать на место в топ-25. позиция в таблице лидеров Kaggle Dogs vs. Cats с использованием архитектуры AlexNet.

5.6 Цветы-17

Набор данных Flowers-17 представляет собой набор данных из 17 категорий с 80 изображениями на класс, созданный Nilsback et al. [52]. Цель этого набора данных — правильно предсказать вид цветка для заданного входного изображения. Образец набора данных Flowers-17 можно увидеть на рис. 5.6.

Цветы-17 можно считать сложным набором данных из-за резких изменений масштаба, углов обзора, фоновых помех, различных условий освещения и различий внутри класса. Кроме того, имея всего 80 изображений на класс, моделям глубокого обучения становится сложно изучить представление для каждого класса без переобучения. Как правило, при обучении глубокой нейронной сети рекомендуется иметь от 1000 до 5000 примеров изображений на класс [10].

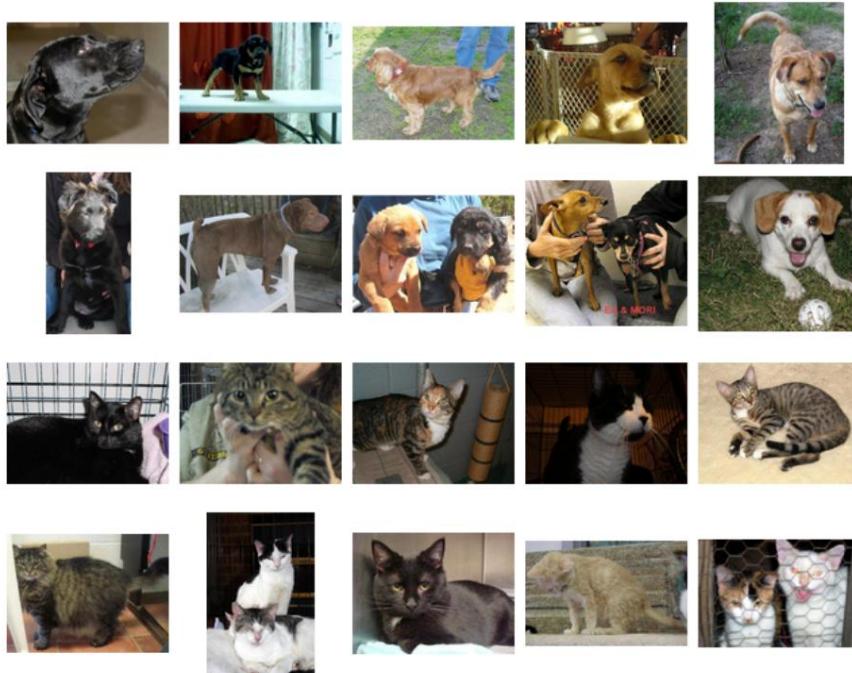


Рисунок 5.5: Образцы из соревнования Kaggle Dogs vs. Cats. Цель этой задачи 2-х классов состоит в том, чтобы правильно идентифицировать данное входное изображение как содержащее «собаку» или «кошку».

Мы изучим набор данных Flowers-17 в пакете Practitioner Bundle и изучим методы улучшения классификации с использованием методов трансферного обучения, таких как извлечение признаков и точная настройка.

5.7 КАЛТЕХ-101

Представлено Fei-Fei et al. [53] В 2004 году набор данных CALTECH-101 стал популярным эталонным набором данных для обнаружения объектов. Обычно используемый для обнаружения объектов (т. е. для прогнозирования (x, y) -координат ограничивающей рамки для конкретного объекта на изображении), мы можем использовать CALTECH-101 также для изучения алгоритмов глубокого обучения.

Набор данных из 8677 изображений включает 101 категорию, охватывающую широкий спектр объектов, включая слонов, велосипеды, футбольные мячи и даже человеческий мозг, и это лишь некоторые из них. Набор данных CALTECH-101 демонстрирует сильный дисбаланс классов (это означает, что для некоторых категорий есть больше примеров изображений, чем для других), что делает его интересным для изучения с точки зрения дисбаланса классов.

Предыдущие подходы к классификации изображений CALTECH-101 получали точность в диапазоне 35-65% [54, 55, 56]. Однако, как я продемонстрирую в пакете Practitioner Bundle, нам легко использовать глубокое обучение для классификации изображений, чтобы получить точность классификации более 99%.

5.8 Крошечный ImageNet 200

В Стенфордском отличном классе cs231n: Convolutional Neural Networks for Visual Recognition [57] собрана задача классификации изображений для студентов, аналогичная задаче ImageNet, но меньшего масштаба. Всего в этом наборе данных 200 классов изображений: 500 изображений для обучения, 50 изображений для проверки и 50 изображений для тестирования на класс. Каждое изображение было предварительно обработано и обрезано до размера $64 \times 64 \times 3$ пикселя, что позволяет учащимся сосредоточиться на методах глубокого обучения, а не на функциях предварительной обработки компьютерного зрения.



Рисунок 5.6: Выборка из пяти (из семнадцати) классов в наборе данных Flowers-17, где каждый класс представляет определенный вид цветов.

Однако, как мы увидим в пакете Practitioner Bundle, этапы предварительной обработки, примененные Карпати и Джонсоном, на самом деле немного усложняют задачу, поскольку некоторая важная, различающая информация вырезается во время задачи предварительной обработки. Тем не менее, я буду демонстрировать, как обучать архитектуры VGGNet, GoogLeNet и ResNet на этом наборе данных и претендовать на первое место в таблице лидеров.

5.9 Аудитория

Набор данных Adience, созданный Eidinger et al. 2014 [58], используется для облегчения изучения распознавания возраста и пола. Всего в набор данных включено 26 580 изображений возрастом от 0 до 60 лет. Цель этого набора данных — правильно предсказать возраст и пол субъекта на изображении. Далее мы обсудим набор данных Adience (и создадим собственные системы распознавания возраста и пола) в пакете ImageNet. Вы можете увидеть образец набора данных Adience на рис. 5.7.

5.10 Имиджнет

В сообществах компьютерного зрения и глубокого обучения вы можете столкнуться с некоторой контекстуальной путаницей в отношении того, что такое ImageNet, а что нет.

5.10.1 Что такое ImageNet?

ImageNet на самом деле является проектом, направленным на маркировку и классификацию изображений почти по 22 000 категорий на основе определенного набора слов и фраз.

На момент написания этой статьи в проекте ImageNet насчитывается более 14 миллионов изображений. Чтобы организовать такое огромное количество данных, ImageNet следует иерархии WordNet [59]. Каждое осмысленное слово/фраза внутри WordNet называется «набором синонимов» или, для краткости, «синнабором». В рамках проекта ImageNet изображения организованы в соответствии с этими синсетами, с целью иметь более 1000 изображений на синсет.

5.10.2 Крупномасштабная программа визуального распознавания ImageNet (ILSVRC)

В контексте компьютерного зрения и глубокого обучения, всякий раз, когда вы слышите, как люди говорят об ImageNet, они, скорее всего, имеют в виду задачу ImageNet Large Scale Visual Recognition Challenge [42],

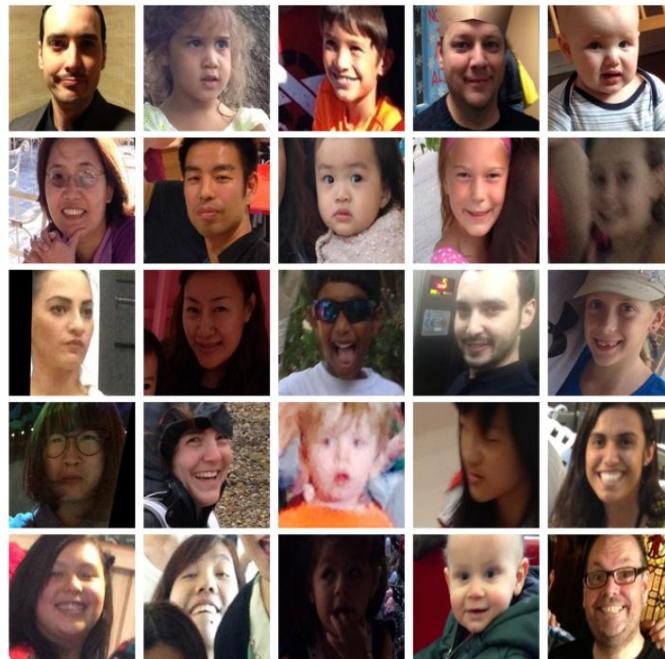


Рисунок 5.7: Пример набора данных Adience для распознавания возраста и пола. Возрастной диапазон от 0 до 60+.

или просто ILSVRC для краткости.

Целью трека классификации изображений в этом задании является обучение модели, которая может классифицировать изображение по 1000 отдельных категорий, используя примерно 1,2 миллиона изображений для обучения, 50 000 для проверки и 100 000 для тестирования. Эти 1000 категорий изображений представляют собой классы объектов, с которыми мы сталкиваемся в повседневной жизни, например, виды собак, кошек, различные предметы домашнего обихода, типы транспортных средств и многое другое. Вы можете найти полный список категорий объектов в вызове ILSVRC здесь: <http://pyimg.co/x1ler>.

Когда дело доходит до классификации изображений, задача ImageNet является стандартом де-факто для алгоритмов классификации компьютерного зрения, и с 2012 года в таблице лидеров этой задачи доминируют сверточные нейронные сети и методы глубокого обучения.

Внутри ImageNet Bundle я покажу, как с нуля обучать оригинальные сетевые архитектуры (AlexNet, SqueezeNet, VGGNet, GoogLeNet, ResNet) на этом популярном наборе данных, что позволит вам воспроизвести самые современные результаты, которые вы видите в соответствующие исследовательские работы.

5.11 Kaggle: задача распознавания выражений лица

Еще одна задача, созданная Kaggle, цель конкурса по распознаванию выражений лица (FER) — правильно определить эмоцию, которую испытывает человек, просто по изображению его лица. В общей сложности 35 888 изображений предоставлены в испытании FER с целью пометить данное выражение лица в семи различных категориях: 1. Злость 2. Отвращение (иногда группируется со «Страхом» из-за дисбаланса классов)

3. Страх

4. Счастливый

5. Грустный 6.

Удивление



Рисунок 5.8: Коллаж из примеров ImageNet, собранных Стенфордским университетом. Этот массив данных содержит более 1,2 миллиона изображений и 1000 возможных категорий объектов. ImageNet считается стандартом де-факто для сравнительного анализа алгоритмов классификации изображений.

7. Нейтральный

Я покажу, как использовать этот набор данных для распознавания эмоций в ImageNet Bundle.

5.12 Внутренний CVPR

Набор данных Indoor Scene Recognition [60], как следует из названия, состоит из ряда внутренних сцен, включая магазины, дома, места отдыха, рабочие зоны и общественные места. Цель этого набора данных — правильно обучить модель, которая может распознавать каждую из областей. Однако вместо того, чтобы использовать этот набор данных по его первоначальному назначению, мы будем использовать его внутри пакета ImageNet для автоматического определения и исправления ориентации изображения.

5.13 Стенфордские автомобили

Другой набор данных, составленный Стенфордом, Cars Dataset [61] состоит из 16 185 изображений 196 классов автомобилей. Вы можете нарезать этот набор данных любым удобным для вас способом, основываясь на марке автомобиля, модели или даже где выпуска. Несмотря на то, что изображений для каждого класса относительно немного (с сильным дисбалансом классов), я продемонстрирую, как использовать сверточные нейронные сети для получения точности классификации > 95% при маркировке марки и модели транспортного средства.

5.14 Резюме

В этой главе мы рассмотрели наборы данных, с которыми вы столкнетесь в оставшейся части книги «Глубокое обучение для компьютерного зрения с помощью Python». Некоторые из этих наборов данных считаются «игрушечными» наборами данных, небольшими наборами изображений, которые мы можем использовать для изучения нейронных сетей и глубокого обучения. Другие наборы данных популярны по историческим причинам и служат отличным эталоном для оценки новых архитектур моделей. Наконец, наборы данных, такие как ImageNet, по-прежнему остаются открытыми темами для исследований и используются для продвижения передовых технологий глубокого обучения.

Потратите время, чтобы кратко ознакомиться с этими наборами данных — я буду обсуждать каждый из них. наборы данных подробно, когда они впервые представлены в соответствующих главах.



Рисунок 5.9: Образец выражений лица в Kaggle: Facial Expression Recognition Challenge. Мы обучим CNN распознавать и идентифицировать каждую из этих эмоций. Эта CNN также сможет работать в режиме реального времени на вашем процессоре, что позволит вам распознавать эмоции в видеопотоках.



Рисунок 5.10: Набор данных Stanford Cars состоит из 16 185 изображений с 196 марками и классами моделей автомобилей. Мы узнаем, как получить точность классификации $> 95\%$ для этого набора данных в пакете ImageNet.

6. Настройка среды разработки

Когда дело доходит до изучения новой технологии (особенно глубокого обучения), настройка среды разработки, как правило, является половиной дела. Из-за разных операционных систем, разных версий зависимостей и самих библиотек настройка собственной среды разработки для глубокого обучения может стать довольно головной болью.

Все эти проблемы усугубляются скоростью, с которой обновляются и выпускаются библиотеки глубокого обучения — новые функции подталкивают к инновациям, но также ломают предыдущие версии. В частности, отличным примером является CUDA Toolkit: в среднем каждый год выходит 2-3 новых выпуска CUDA. год.

С каждым новым выпуском появляются оптимизации, новые функции и возможность быстрее обучать нейронные сети. Но каждый выпуск еще больше усложняет обратную совместимость. Этот быстрый цикл выпуска подразумевает, что глубокое обучение зависит не только от того, как вы настроили среду разработки, но и от того, когда вы ее настроили. В зависимости от временных рамок ваша среда может устареть!

Из-за быстро меняющегося характера зависимостей и библиотек глубокого обучения я решил переместить большую часть этой главы на вспомогательный веб-сайт (<http://dl4cv.pyimagesearch.com/>) , чтобы новые, свежие руководства всегда были доступны для вас использовать.

Вам следует использовать эту главу, чтобы ознакомиться с различными библиотеками глубокого обучения , которые мы будем использовать в этой книге, а затем следовать инструкциям на страницах со ссылками на эти библиотеки из этой книги.

6.1 Библиотеки и пакеты

Чтобы стать успешным практиком глубокого обучения, нам нужен правильный набор инструментов и пакетов. В этом разделе подробно описывается язык программирования, а также основные библиотеки, которые мы будем использовать для изучения глубокого обучения для компьютерного зрения.

6.1.1 Python .

Мы будем использовать язык программирования Python для всех примеров в Deep Learning for Computer Vision with Python. Python — это простой язык для изучения, и он, безусловно, лучший

способ работы с алгоритмами глубокого обучения. Простой, интуитивно понятный синтаксис позволяет вам сосредоточиться на изучении основ глубокого обучения, а не тратить часы на исправление сумасшедших ошибок компилятора в других языках.

6.1.2 Керас

Для создания и обучения наших сетей глубокого обучения мы в первую очередь будем использовать библиотеку Keras. Keras поддерживает как TensorFlow, так и Theano, что упрощает создание и быстрое обучение сетей.

Пожалуйста, обратитесь к разделу 6.2 для получения дополнительной информации о совместимости TensorFlow и Theano с Keras.

6.1.3 Мкснет

Мы также будем использовать mxnet, библиотеку глубокого обучения, которая специализируется на распределенном многомашинном обучении. Возможность распараллелить обучение на нескольких графических процессорах/устройствах имеет решающее значение при обучении архитектур глубоких нейронных сетей на массивных наборах данных изображений (таких как ImageNet).

 Библиотека mxnet используется только в пакете ImageNet этой книги.

6.1.4 OpenCV, scikit-image, scikit-learn и другие

Поскольку эта книга посвящена применению глубокого обучения к компьютерному зрению, мы также будем использовать несколько дополнительных библиотек. Вам не нужно быть экспертом в этих библиотеках или иметь опыт работы с ними, чтобы успешно пользоваться этой книгой, но я рекомендую ознакомиться с основами OpenCV, если вы можете. Первых пяти глав «[Практического Python и OpenCV](#)» более чем достаточно для понимания основ библиотеки OpenCV.

Основная цель OpenCV — обработка изображений в реальном времени. Эта библиотека существует с 1999 года, но только в выпуске 2.0 в 2009 году мы увидели невероятную поддержку Python, которая включала представление изображений в виде массивов NumPy.

Сам OpenCV написан на C/C++, но при установке предоставляются привязки Python. OpenCV является стандартом де-факто, когда дело доходит до обработки изображений, поэтому мы будем использовать его при загрузке изображений с диска, отображении их на нашем экране и выполнении основных операций обработки изображений.

Чтобы дополнить OpenCV, мы также будем использовать небольшой фрагмент scikit-image [62] (scikit-image.org), набор алгоритмов обработки изображений.

Scikit-learn [5] (scikit-learn.org) — это библиотека Python с открытым исходным кодом для машинного обучения, перекрестной проверки и визуализации — эта библиотека хорошо дополняет Keras и помогает нам не «изобретать велосипед», особенно когда речь идет о разделении обучения/тестирования/проверки и проверке точности наших моделей глубокого обучения.

6.2 Настройка среды разработки?

Если вы готовы настроить свою среду глубокого обучения, просто нажмите на ссылку ниже и следуйте предоставленным инструкциям для вашей операционной системы и независимо от того, будете ли вы использовать графический процессор:

<http://pyimg.co/k81c6>

 Если вы еще не создали свою учетную запись на сопутствующем веб-сайте для Deep Learning for Computer Vision with Python, см. первые несколько страниц этой книги (сразу после оглавления) для получения ссылки для регистрации. Оттуда создайте свою учетную запись, и вы сможете получить доступ к дополнительным материалам.

6.3 Предварительно настроенная виртуальная машина

Я понимаю, что настройка вашей среды разработки может быть не только трудоемкой и утомительной задачей, но и потенциально серьезным препятствием для входа, если вы новичок в средах на основе Unix. Из-за этой трудности ваша покупка Deep Learning for Computer Vision with Python включает в себя предварительно настроенную виртуальную машину Ubuntu VirtualBox, которая поставляется со всеми необходимыми библиотеками глубокого обучения и компьютерного зрения, которые вам понадобятся для успешного использования этой книги, предварительно настроенной и предустановленной..

Обязательно загрузите файл VirtualMachine.zip, входящий в комплект, чтобы получить доступ к этой виртуальной машине. Инструкции по настройке и использованию вашей виртуальной машины можно найти в файле README.pdf, прилагаемом к этой книге.

6.4 Облачные экземпляры

Основным недостатком виртуальной машины Ubuntu является то, что по самому определению виртуальной машины виртуальной машине не разрешен доступ к физическим компонентам вашей хост-машины (например, к графическому процессору). При обучении больших сетей глубокого обучения наличие графического процессора чрезвычайно полезно.

Для тех, кто хочет иметь доступ к графическому процессору при обучении своих нейронных сетей, я бы предложил: 1. Настройка инстанса Amazon EC2 с поддержкой графического процессора.

2. Зарегистрируйте учетную запись FloydHub и настройте свой экземпляр GPU в облаке.

Важно отметить, что стоимость каждого из этих вариантов зависит от количества часов (EC2) или секунд (FloydHub), в течение которых загружается ваш экземпляр. Если вы решите пойти по пути «GPU в облаке», обязательно сравните цены и помните о своих расходах — нет ничего хуже, чем неожиданно получить крупный счет за использование облака.

Если вы решите использовать облачный экземпляр, я рекомендую вам использовать мой предварительно настроенный экземпляр машины Amazon (AMI). AMI поставляется со всеми библиотеками глубокого обучения, которые вам понадобятся в этой книге, предварительно сконфигурированными и предустановленными.

Дополнительные сведения об AMI см. на сопутствующем веб-сайте Deep Learning for Computer Vision with Python.

6.5 Как структурировать ваши проекты

Теперь, когда у вас появилась возможность настроить среду разработки, найдите секунду и загрузите ZIP-файл с кодом и наборами данных, связанными с Starter Bundle.

После того, как вы загрузили файл, разархивируйте его, и вы увидите следующую структуру каталогов:

```
|--- sb_code |
|--- Chapter07
|   |--- image_classifier | |--- Chapter08-
|   |--- parameterized_learning | |--- Chapter09-optimization_methods
...
|   |--- наборы данных
```

Каждая глава (включая сопровождающий код) имеет свой собственный каталог. Затем каждый каталог включает:

- Исходный код главы.
- Библиотека ruimagesearch для глубокого обучения, которую вы будете создавать по ходу работы.

с книгой.

• Любые дополнительные файлы, необходимые для запуска соответствующих примеров.

Каталог наборов данных , как следует из названия, содержит все наборы данных изображений для Starter Bundle.

В качестве примера предположим, что я хотел обучить свой первый классификатор изображений. Я бы сначала изменил каталог на Chapter07-first_image_classifier , а затем выполнил скрипт knn.py , указав аргумент командной строки --dataset на набор данных животных:

```
$ cd ..//chapter07-first_image_classifier/ $ python  
knn.py --dataset ..//datasets/animals
```

Это даст скрипту knn.py указание обучить простой классификатор k-ближайших соседей (k-NN) на наборе данных «животные» (который является подкаталогом внутри наборов данных), небольшой коллекции собак, кошек и панд на изображениях.

Если вы новичок в командной строке и в том, как использовать аргументы командной строки, я настоятельно рекомендую вам прочитать об аргументах командной строки и о том, как их использовать, прежде чем углубляться в эту книгу:

<http://pyimg.co/vsapz>

Освоение командной строки (и отладка ошибок с помощью терминала)
очень важный навык для вас развивать.

Наконец, в качестве небольшого примечания, я хотел бы упомянуть, что я предпочитаю хранить свои наборы данных отдельно от исходного кода, поскольку это: • сохраняет структуру моего проекта в чистоте и порядке • позволяет мне повторно использовать наборы данных в нескольких проектах. структура каталогов для ваших собственных проектов.

6.6 Резюме

Когда дело доходит до настройки среды разработки для глубокого обучения, у вас есть несколько вариантов. Если вы предпочитаете работать со своего локального компьютера, это вполне разумно, но сначала вам нужно будет скомпилировать и установить некоторые зависимости. Если вы планируете использовать CUDA-совместимый графический процессор на локальном компьютере, также потребуется несколько дополнительных шагов установки.

Для читателей, которые не знакомы с настройкой своей среды разработки, или для читателей, которые просто хотят вообще пропустить этот процесс, обязательно взгляните на предварительно настроенную виртуальную машину Ubuntu VirtualBox, включенную в загружаемый пакет Deep Learning for Computer Vision with Python.

Если вы хотите использовать графический процессор, но он не подключен к вашей системе, рассмотрите возможность использования облачных экземпляров, таких как Amazon EC2 или FloydHub. Хотя за использование этих сервисов взимается почасовая плата, они могут сэкономить вам деньги по сравнению с покупкой графического процессора заранее.

Наконец, имейте в виду, что если вы планируете проводить какие-либо серьезные исследования или разработки в области глубокого обучения, рассмотрите возможность использования среды Linux, такой как Ubuntu. В то время как работу по глубокому обучению можно выполнять в Windows (не рекомендуется) или macOS (абсолютно приемлемо, если вы только начинаете), почти все рабочие среды для глубокого обучения используют операционные системы на базе Linux — помните об этом факте, когда будете настраиваете собственную среду разработки для глубокого обучения.

7. Ваш первый классификатор изображений

За последние несколько глав мы потратили достаточно много времени на обсуждение основ изображений, типов обучения и даже четырехэтапного конвейера, которому мы можем следовать при создании наших собственных классификаторов изображений. Но нам еще предстоит создать собственный классификатор изображений.

В этой главе это изменится. Мы начнем с создания нескольких вспомогательных утилит для облегчения предварительной обработки и загрузки изображений с диска. Оттуда мы обсудим классификатор k-ближайших соседей (k-NN), ваш первый опыт использования машинного обучения для классификации изображений. На самом деле, этот алгоритм настолько прост, что не выполняет никакого фактического «обучения», тем не менее, это важный алгоритм для обзора, чтобы мы могли оценить, как нейронные сети учатся на данных в будущих главах.

Наконец, мы применим наш алгоритм k-NN для распознавания различных видов животных на изображениях.

7.1 Работа с наборами данных изображений

При работе с наборами данных изображений мы в первую очередь должны учитывать общий размер набора данных в байтах. Достаточно ли велик наш набор данных, чтобы поместиться в доступную оперативную память на нашем компьютере? Можем ли мы загрузить набор данных, как если бы загружали большую матрицу или массив? Или набор данных настолько велик, что превышает память нашей машины, что требует от нас «разбивать» набор данных на сегменты и загружать только части за раз?

Все наборы данных внутри Starter Bundle достаточно малы, чтобы мы могли загружать их в основную память, не беспокоясь об управлении памятью; однако гораздо большие наборы данных внутри пакетов Practitioner Bundle и ImageNet Bundle потребуют от нас разработки некоторых умных методов для эффективной обработки загрузки изображений таким образом, чтобы мы могли обучить классификатор изображений (без нехватки памяти).

Тем не менее, вы всегда должны знать размер вашего набора данных, прежде чем даже начать работать с алгоритмами классификации изображений. Как мы увидим в оставшейся части этой главы, время, потраченное на организацию, предварительную обработку и загрузку набора данных, является критическим аспектом построения классификатора изображений.

7.1.1 Знакомство с набором данных «Животные»

Набор данных «Животные» — это простой пример набора данных, который я собрал, чтобы продемонстрировать, как обучать классификаторы изображений с использованием простых методов машинного обучения, а также передовых алгоритмов глубокого обучения.

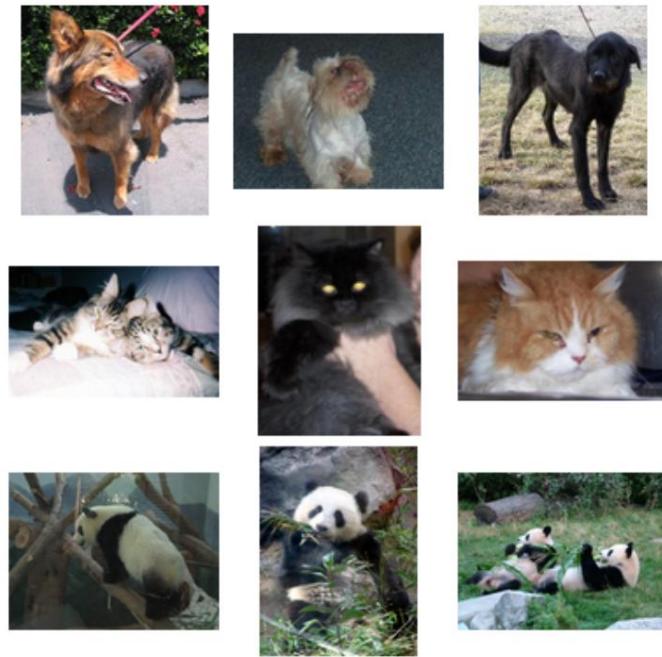


Рисунок 7.1: Выборка набора данных о трех классах животных, состоящего из 1000 изображений для каждого класса собак, кошек и панд соответственно, всего 3000 изображений.

Изображения в наборе данных *Animals* относятся к трем различным классам: собаки, кошки и панды, по 1000 примеров изображений на класс. Изображения собак и кошек были взяты из конкурса Kaggle Dogs vs. Cats (<http://pyimg.co/ogx37>) , а изображения панд были взяты из набора данных ImageNet [42].

Набор данных *Animals*, содержащий всего 3000 изображений, может легко поместиться в основную память наших машин, что значительно ускорит обучение наших моделей, не требуя от нас написания какого-либо «служебного кода» для управления набором данных, который иначе не мог бы поместиться в память. Лучше всего то, что модель глубокого обучения можно быстро обучить на этом наборе данных как на ЦП, так и на графическом процессоре. Независимо от конфигурации вашего оборудования вы можете использовать этот набор данных для изучения основ машинного обучения и глубокого обучения.

Наша цель в этой главе — использовать классификатор k-NN, чтобы попытаться распознать каждый из этих видов на изображении, используя только необработанные интенсивности пикселей (т. е. без выделения признаков). Как мы увидим, необработанные интенсивности пикселей плохо подходят для алгоритма k-NN. Тем не менее, это важный контрольный эксперимент, который необходимо провести, чтобы мы могли понять, почему сверточные нейронные сети могут получать такую высокую точность в отношении интенсивности необработанных пикселей, в то время как традиционные алгоритмы машинного обучения не могут этого сделать.

7.1.2 Начало нашего набора инструментов для глубокого обучения

Как я упоминал в разделе 1.5, на протяжении всей этой книги мы будем создавать собственный инструментарий глубокого обучения . Мы начнем с базовых вспомогательных функций и классов для предварительной обработки изображений и загрузки небольших наборов данных, в конечном итоге достроившись до реализаций современных сверточных нейронных сетей.

Фактически, это тот же набор инструментов, который я использую при проведении собственных экспериментов по глубокому обучению. Этот инструментарий будет создаваться по частям, глава за главой, что позволит вам увидеть отдельные компоненты, составляющие пакет, и в конечном итоге станет полноценной библиотекой, которую можно использовать для быстрого создания и обучения ваших собственных сетей глубокого обучения.

Давайте продолжим и начнем определять структуру проекта нашего инструментария:

7.1 Работа с наборами данных изображений

```
|--- pyimagesearch
```

Как видите, у нас есть единственный модуль с именем `pyimagesearch`. Весь код, который мы разрабатываем, будет существовать внутри модуля `pyimagesearch`. Для целей этой главы нам потребуется определить два подмодуля:

```
|--- pyimagesearch
| |--- __init__.py
| |--- наборы данных
| | |--- __init__.py
| | |--- simpledatasetloader.py
| |--- предварительная обработка
| | |--- __init__.py
| | |--- simplepreprocessor.py
```

Подмодуль наборов данных запустит нашу реализацию класса `SimpleDatasetLoader`.

Мы будем использовать этот класс для загрузки небольших наборов данных изображений с диска (которые могут поместиться в основную память), предварительно обработать каждое изображение в наборе данных в соответствии с набором функций, а затем вернуть:

1. Изображения (т. е. исходная интенсивность пикселей)
2. Метка класса, связанная с каждым изображением

Затем у нас есть подмодуль предварительной обработки. Как мы увидим в следующих главах, существует множество методов предварительной обработки, которые мы можем применить к нашему набору данных изображений, чтобы улучшить классификацию точность, включая вычитание среднего, выборку случайных участков или просто изменение размера изображения до исправленный размер. В этом случае наш класс `SimplePreprocessor` сделает последнее — загрузит изображение из диска и изменил его размер до фиксированного размера, игнорируя соотношение сторон. В следующих двух разделах мы реализуем `SimplePreprocessor` и `SimpleDatasetLoader` вручную.

 Пока мы будем рассматривать весь модуль `pyimagesearch` для глубокого обучения в этом книгу я намеренно оставил объяснения файлов `__init__.py` в качестве упражнения для читателя. Эти файлы просто содержат ярлыки импорта и не имеют отношения к пониманию глубоких методов обучения и машинного обучения, применяемые к классификации изображений. Если вы новичок в язык программирования Python, я бы посоветовал освежить в памяти основы импорт [63] (<http://pyimg.co/7w238>).

7.1.3 Базовый препроцессор изображений

Алгоритмы машинного обучения, такие как k-NN, SVM и даже сверточные нейронные сети. требуют, чтобы все изображения в наборе данных имели фиксированный размер вектора признаков. В случае изображений это требование подразумевает, что наши изображения должны быть предварительно обработаны и масштабированы, чтобы иметь одинаковую ширину и высоты.

Существует несколько способов выполнить это изменение размера и масштабирования, начиная от более сложных методов, учитывающие соотношение сторон исходного изображения и масштабированного изображения, до простых методов, игнорируйте соотношение сторон и просто уменьшайте ширину и высоту до требуемых размеров. Точно какой метод вам следует использовать, зависит от сложности ваших факторов вариации (раздел 4.1.3) — в некоторых случаях игнорирование соотношения сторон работает просто отлично; в других случаях вы захотите сохранить соотношение сторон.

В этой главе мы начнем с базового решения: создания препроцессора изображений, изображение, игнорируя соотношение сторон. Откройте `simplepreprocessor.py` и вставьте следующий код:

```

1 # импортируем необходимые пакеты
2 import cv2
3
4 класс SimplePreprocessor:
5     def __init__(я, ширина, высота, интер=cv2.INTER_AREA):
6         # сохранить ширину, высоту и интерполяцию целевого изображения
7         # метод, используемый при изменении размера
8         собственная ширина = ширина
9
10        self.height = высота
11        self.inter = интер
12
13    предварительная обработка def (я, изображение):
14        # изменить размер изображения до фиксированного размера, игнорируя аспект
15        # соотношение
16        вернуть cv2.resize (изображение, (self.width, self.height),
17                           интерполяция = self.inter)

```

Строка 2 импортирует наш единственный необходимый пакет, наши привязки OpenCV. Затем мы определяем конструктор в класс SimpleProcessor в строке 5. Конструктор требует два аргумента, за которыми следует третий необязательный, каждый из которых подробно описан ниже:

- ширина: Целевая ширина нашего входного изображения после изменения размера.
- высота: целевая высота нашего входного изображения после изменения размера.
- inter: необязательный параметр, используемый для управления тем, какой алгоритм интерполяции используется при изменение размера.

Функция предварительной обработки определена в строке 12 и требует единственного аргумента — входного изображения .

которые мы хотим предварительно обработать.

Строки 15 и 16 предварительно обрабатывают изображение, изменяя его размер до фиксированных размеров ширины и высоты , которые затем мы возвращаемся к вызывающей функции.

Опять же, этот препроцессор по определению очень простой — все, что мы делаем, это принимаем входные данные. изображение, изменяя его размер до фиксированного размера, а затем возвращая его. Однако в сочетании с загрузчик набора данных изображений в следующем разделе, этот препроцессор позволит нам быстро загружать и предварительная обработка набора данных с диска, что позволяет нам быстро перемещаться по конвейеру классификации изображений. и перейти к более важным аспектам, таким как обучение нашего фактического классификатора.

7.1.4 Создание загрузчика изображений

Теперь, когда наш SimplePreprocessor определен, давайте перейдем к SimpleDatasetLoader:

```

1 # импортируем необходимые пакеты
2 импортировать numpy как np
3 импорт cv2
4 импорт ОС
5
6 класс SimpleDatasetLoader:
7     def __init__(я, препроцессоры=None):
8         # сохраняем препроцессор изображения
9         self.preprocessors = препроцессоры
10
11     # если препроцессоры None, инициализировать их как
12     # пустой список
13     если self.preprocessors равен None:
14         self.preprocessors = []

```

Строки 2-4 импортируют наши необходимые пакеты Python: NumPy для числовой обработки, cv2 для нашего Привязки OpenCV и os, чтобы мы могли извлекать имена подкаталогов в путях к изображениям.

Строка 7 определяет конструктор для `SimpleDatasetLoader`, где мы можем опционально передать список препроцессоров изображений (таких как `SimpleProcessor`), которые можно последовательно применять к данному входное изображение.

Указание этих препроцессоров в виде списка, а не одного значения, важно — раз, когда нам сначала нужно изменить размер изображения до фиксированного размера, а затем выполнить какое-то масштабирование (например, как вычитание среднего), с последующим преобразованием массива изображений в формат, подходящий для Keras. Каждый из этих препроцессоров могут быть реализованы независимо друг от друга, что позволяет нам применять их последовательно к изображению эффективным образом.

Затем мы можем перейти к методу загрузки, ядру `SimpleDatasetLoader`:

```

16     def load(self, imagePaths, verbose=-1):
17         # инициализируем список функций и меток
18         данные = []
19         метки = []
20
21         # цикл по входным изображениям
22         для (i, imagePath) в перечислении (imagePaths):
23             # загрузите изображение и извлеките метку класса, предполагая, что
24             # что наш путь имеет следующий формат:
25             # /путь/к/набору/данных/{класс}/{изображение}.jpg
26             изображение = cv2.imread(путь к изображению)
27             label = imagePath.split(os.path.sep)[-2]

```

Для нашего метода загрузки требуется один параметр — `imagePaths`, представляющий собой список, указывающий пути к файлам изображений в нашем наборе данных, находящихся на диске. Мы также можем предоставить значение для `verbose`. Этот «уровень детализации» можно использовать для печати обновлений на консоли, что позволяет нам отслеживать, сколько изображения, обработанные `SimpleDatasetLoader`.

Строки 18 и 19 инициализируют наш список данных (то есть самих изображений) вместе с метками, список меток классов для наших изображений.

В строке 22 мы начинаем перебирать каждое из входных изображений. Для каждого из этих изображений мы загружаем его с диска (строка 26) и извлеките метку класса на основе пути к файлу (строка 27). Мы делаем Предположим, что наши наборы данных организованы на диске в соответствии со следующей структурой каталогов:

```
/dataset_name/class/image.jpg
```

`Имя_набора_данных` может быть любым именем набора данных, в данном случае животные.

`class` должно быть именем метки класса. В нашем примере класс — собака, кошка или панда.

Наконец, `image.jpg` — это имя самого изображения.

Основываясь на этой иерархической структуре каталогов, мы можем содержать наши наборы данных в чистоте и порядке. Этого Таким образом, можно с уверенностью предположить, что все изображения внутри подкаталога `dog` являются примерами собак. Сходным образом, мы предполагаем, что все изображения в каталоге `pand` содержат примеры панд.

Почти каждый набор данных, который мы просматриваем в *Deep Learning for Computer Vision with Python*, будет следуйте этой иерархической структуре каталогов — я настоятельно рекомендую вам сделать то же самое для ваши собственные проекты, а также.

Теперь, когда наше изображение загружено с диска, мы можем предварительно обработать его (при необходимости):

```

29     # проверяем, не являются ли наши препроцессоры None
30     если self.preprocessors не None:

```

```

31         # перебираем препроцессоры и применяем каждый к
32         # Изображение
33         для p в self.preprocessors:
34             изображение = p.preprocess(изображение)
35
36         # рассматриваем наше обработанное изображение как "вектор признаков"
37         # путем обновления списка данных, за которым следуют метки
38         data.append(изображение)
39         labels.append(метка)

```

В строке 30 выполняется быстрая проверка, чтобы убедиться, что наши препроцессоры не равны `None`. Если чек проходит, мы перебираем каждый из препроцессоров в строке 33 и последовательно применяем их к изображению на строке 34 — это действие позволяет нам сформировать цепочку препроцессоров, которые можно применить к каждое изображение в наборе данных.

После предварительной обработки изображения мы обновляем списки данных и меток соответственно (строки 39 и 39).

Наш последний блок кода просто выполняет печать обновлений на нашу консоль, а затем возвращает 2-кортеж данных и меток вызывающей функции:

```

41     # показывать обновление для каждого «подробного» изображения
42     если verbose > 0 и i > 0 и (i + 1) % verbose == 0:
43         print("[INFO] обработано {}/{}".format(i + 1,
44                                         len(пути к изображениям)))
45
46     # вернуть кортеж данных и меток
47     возврат (np.array(данные), np.array(метки))

```

Как видите, наш загрузчик набора данных прост по своей конструкции; тем не менее, это дает нам возможность с легкостью применять любое количество процессоров изображений к каждому изображению в нашем наборе данных. Единственное предостережение от этого загрузчика набора данных заключается в том, что он предполагает, что все изображения в наборе данных могут одновременно помещаться в основную память.

Для наборов данных, которые слишком велики, чтобы поместиться в ОЗУ вашей системы, нам потребуется разработать более сложный загрузчик наборов данных — я расскажу об этих более продвинутых загрузчиках наборов данных в пакете Practitioner Bundle. Теперь, когда мы понимаем, как (1) предварительно обработать изображение и (2) загрузить коллекцию изображений из диска, теперь мы можем перейти к этапу классификации изображений.

7.2 k-NN: простой классификатор

Классификатор k-Nearest Neighbor на сегодняшний день является самым простым алгоритмом машинного обучения и классификации изображений. На самом деле, это настолько просто, что на самом деле ничему не «обучается». Вместо этого этот алгоритм напрямую зависит от расстояния между векторами признаков (которые в нашем случае являются необработанными интенсивности пикселей RGB изображений).

Проще говоря, алгоритм k-NN классифицирует неизвестные точки данных, находя наиболее распространенные. класс среди k ближайших примеров. Каждая точка данных из k ближайших точек данных дает право голоса, и категория с наибольшим количеством голосов побеждает. Или, говоря простым языком: «Скажи мне, кто твои соседи? есть, и я скажу вам, кто вы» [64], как показано на рис. 7.2.

Чтобы алгоритм k-NN работал, он делает основное предположение, что изображения с подобные визуальные содержания лежат близко друг к другу в n -мерном пространстве. Здесь мы видим три категории изображений, обозначенные как собаки, кошки и панды соответственно. В этом притворном примере мы отложили «пушистость» шерсти животного по оси абсцисс и «легкость» шерсти. пальто вдоль оси Y. Каждая из точек данных животных сгруппирована относительно близко друг к другу в нашем

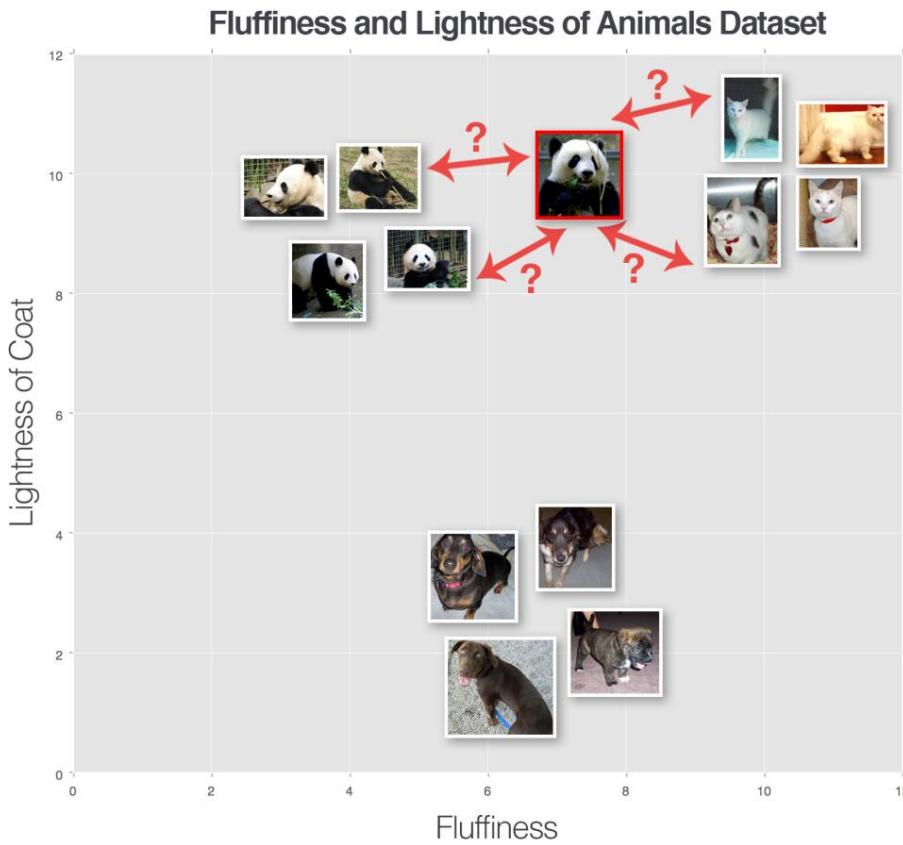


Рисунок 7.2. Учитывая наш набор данных о собаках, кошках и пандах, как мы можем классифицировать изображение, обведенное красным?

п-мерное пространство. Это означает, что расстояние между двумя изображениями кошек намного меньше, чем расстояние между кошкой и собакой.

Однако, чтобы применить классификатор k-NN, нам сначала нужно выбрать метрику расстояния или функцию сходства. Обычный выбор включает евклидово расстояние (часто называемое L2-расстоянием):

$$d(p,q) = \sqrt{\sum_{i=1}^n (p_i - q_i)^2} \quad (7.1)$$

Однако другие показатели расстояния, такие как Манхэттен/городской квартал (часто называемый расстоянием L1) также можно использовать:

$$d(p,q) = \sum_{i=1}^n |p_i - q_i| \quad (7.2)$$

На самом деле вы можете использовать любую функцию метрики/сходства расстояния, которая больше всего подходит для ваших данных (и дает вам наилучшие результаты классификации). Однако до конца этого урока мы будем использовать самую популярную метрику расстояния: евклидово расстояние.

7.2.1 Пример работы k-NN

На данный момент мы понимаем принципы алгоритма k-NN. Мы знаем, что классификация зависит от расстояния между векторами признаков/изображениями. И мы знаем, что для вычисления этих расстояний требуется функция расстояния/подобия.

Но как мы на самом деле делаем классификацию? Чтобы ответить на этот вопрос, давайте посмотрим на рисунок 7.3. Здесь у нас есть набор данных о трех типах животных — собаках, кошках и пандах — и мы нанесли их на график в соответствии с их пушистостью и легкостью шерсти.

Мы также вставили «неизвестное животное», которое пытаемся классифицировать, используя только одного соседа (т. е. $k = 1$). В этом случае ближайшее животное к входному изображению является точкой данных собаки; таким образом, наше входное изображение должно быть классифицировано как собака.

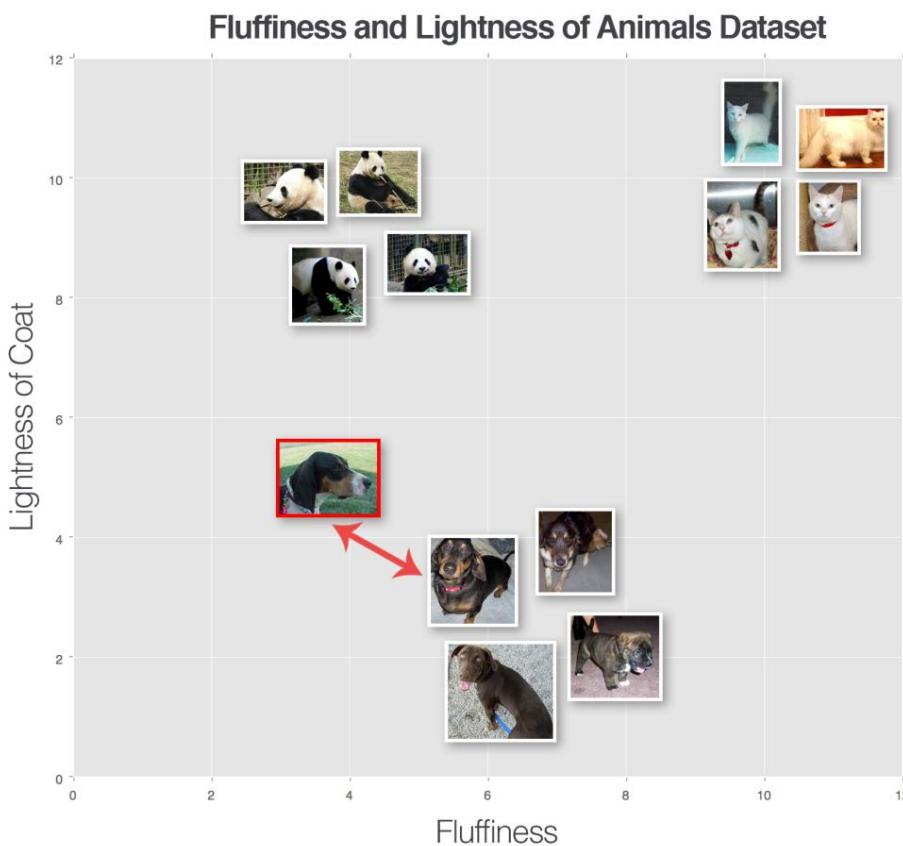


Рисунок 7.3: В этом примере мы вставили неизвестное изображение (выделено красным) в набор данных, а затем использовали расстояние между неизвестным животным и набором данных животных, чтобы сделать классификацию.

Давайте попробуем другое «неизвестное животное», на этот раз используя $k = 3$ (рис. 7.4). Мы нашли двух кошек и одну панду в тройке лучших результатов. Поскольку категория кошек имеет наибольшее количество голосов, мы классифицируем наше входное изображение как кошку.

Мы можем продолжать выполнять этот процесс для различных значений k , но независимо от того, насколько большим или малым становится k , принцип остается тем же — побеждает категория с наибольшим количеством голосов в k ближайших обучающих точках, которая используется в качестве метки для точки входных данных.

 В случае ничьей алгоритм k-NN случайным образом выбирает одну из связанных меток классов.

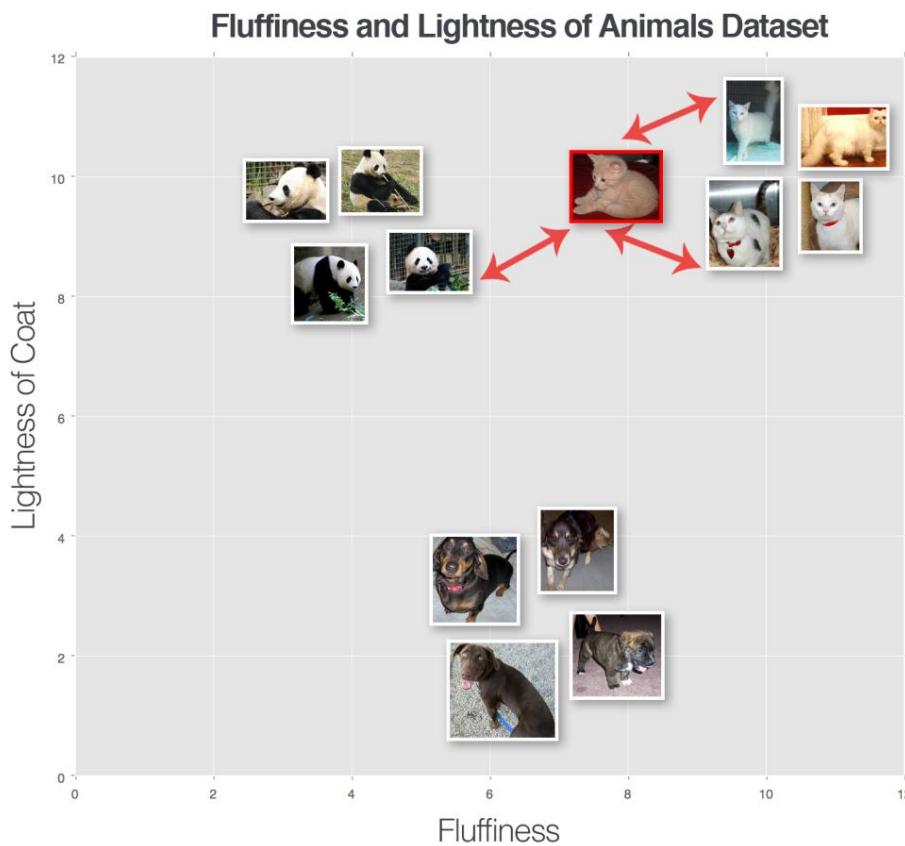


Рисунок 7.4: Классификация другого животного, только на этот раз мы использовали $k = 3$, а не просто $k = 1$. Поскольку есть два изображения кошек ближе к входному изображению, чем одно изображение панды, мы пометим это входное изображение как кошку.

7.2.2 Гиперпараметры k-NN

Есть два четких гиперпараметра, которые нас интересуют при запуске алгоритма k-NN. Первое очевидно: значение k . Каково оптимальное значение k ? Если он слишком мал (например, $k = 1$), мы получаем эффективность, но становимся восприимчивы к шуму и выбросам данных. Однако, если k слишком велико, мы рискуем чрезмерно сгладить результаты нашей классификации и увеличить погрешность.

Второй параметр, который мы должны учитывать, — это фактическая метрика расстояния. Является ли евклидово расстояние лучшим выбором? А как насчет манхэттенского расстояния?

В следующем разделе мы обучим наш классификатор k-NN на наборе данных Animals и оценим модель на нашем тестовом наборе. Я бы посоветовал вам поиграть с разными значениями k вместе с различными показателями расстояния, отмечая, как меняется производительность.

Исчерпывающий обзор того, как настроить гиперпараметры k-NN, см. в уроке 4.3. внутри курса PyImageSearch Gurus [33].

7.2.3 Реализация k-NN Целью этого раздела

является обучение классификатора k-NN на необработанных интенсивностях пикселей набора данных Animals и использование его для классификации неизвестных изображений животных. Мы будем использовать наш четырехступенчатый конвейер для обучения классификаторов из раздела 4.3.2:

- Шаг № 1. Соберите наш набор данных. Наборы данных «Животные» состоят из 3000 изображений, по 1000 изображений для собак, кошек и классов панд соответственно. Каждое изображение представлено в RGB

цветовое пространство. Мы предварительно обрабатываем каждое изображение, изменив его размер до 32×32 пикселей.

Принимая во внимание три канала RGB, размеры изображения с измененным размером подразумевают, что каждое изображение в наборе данных представлено $32 \times 32 \times 3 = 3072$ целыми числами.

- Шаг №2 – Разделение набора данных. В этом простом примере мы будем использовать два разделения данных.

Один сплит для обучения, а другой для тестирования. Мы опустим проверочный набор для настройки гиперпараметров и оставим это читателю в качестве упражнения.

- Шаг № 3. Обучение классификатора. Наш классификатор k-NN будет обучаться на необработанных интенсивностях пикселей. связи изображений в обучающей выборке.
- Шаг № 4 — Оценка: как только наш классификатор k-NN обучен, мы можем оценить производительность на тестовый набор.

Давайте идти вперед и начать. Откройте новый файл, назовите его knn.py и вставьте следующий код:

```
1 # импортируем необходимые пакеты 2
из sklearn.neighbors импортируем KNeighborsClassifier 3 из
sklearn.preprocessing импортируем LabelEncoder 4 из
sklearn.model_selection импортируем train_test_split 5 из sklearn.metrics
----- пути импорта imutils 9 импорт argparse
```

Строки 2-9 импортируют необходимые пакеты Python. Наиболее важные из них, на которые следует обратить внимание:

Строка 2: KNeighborsClassifier — это наша реализация алгоритма k-NN, про предоставлено библиотекой scikit-learn.

- Страна 3: LabelEncoder, вспомогательная утилита для преобразования меток, представленных в виде строк, в целые числа, где имеется одно уникальное целое число для каждой метки класса (обычная практика при применении машинного обучения).
- Страна 4: мы импортируем функцию train_test_split, которая представляет собой удобную вспомогательную функцию, помогающую нам создавать сплиты для обучения и тестирования.
- Страна 5: функция classification_report — еще одна служебная функция, которая помогает нам оценить производительность нашего классификатора и вывести на консоль хорошо отформатированную таблицу результатов.

Вы также можете увидеть наши реализации SimplePreprocessor и SimpleDatasetLoader . импортируется в строки 6 и 7 соответственно.

Далее, давайте проанализируем наши аргументы командной строки:

```
11 # построить разбор аргумента и разобрать аргументы 12 ap =
argparse.ArgumentParser() 13 ap.add_argument("-d", "--dataset",
required=True,
14     help="путь к входному набору
15 ap.add_argument("-k", "-neighbours", type=int, default=1, help="
16         количество ближайших соседей для классификации")
17 ap.add_argument("-j", "-jobs", type=int, default=-1, help=" количество
18         заданий для расстояния k-NN (-1 использует все доступные ядра)") 19 args = vars
(ap.parse_args())
```

Наш скрипт требует один аргумент командной строки, за которым следуют два необязательных аргумента, каждый из которых рассмотрен ниже:

• `--dataset`: Путь к месту на диске, где находится наш входной набор данных изображений. • `--neighbours`: Необязательный параметр, количество соседей k , применяемых при использовании алгоритма k-NN. • `--jobs`: необязательно, количество одновременных заданий для выполнения при вычислении расстояния между точкой входных данных и обучающим набором. Значение -1 будет использовать все доступные ядра процессора.

Теперь, когда наши аргументы командной строки проанализированы, мы можем получить пути к файлам изображений в наш набор данных с последующей их загрузкой и предварительной обработкой (шаг № 1 в конвейере классификации):

```

21 # получить список изображений, которые мы будем
описывать 22 print("[INFO] загрузка изображений...") 23
imagePaths = list(paths.list_images(args["dataset"])) 24
25 # инициализировать препроцессор изображений, загрузить набор данных
с диска, 26 # и изменить форму матрицы данных 27 sp = SimplePreprocessor(32,
32) 28 sdl = SimpleDatasetLoader(preprocessors=[sp]) 29 (data, labels) = sdl.load
(imagePaths, verbose=500) 30 data = data.reshape((data.shape[0], 3072))
31
32 # показать некоторую информацию о потреблении памяти
изображениями 33 print("[INFO] features matrix: {:.1f}MB".format( data.nbytes /
34 (1024 * 1000.0)))

```

Строка 23 получает пути ко всем изображениям в нашем наборе данных. Затем мы инициализируем наш `SimplePreprocessor`, используемый для изменения размера каждого изображения до 32×32 пикселей в строке 27.

`SimpleDatasetLoader` инициализируется в строке 28, предоставляя созданный экземпляр `SimplePreprocessor`. В качестве аргумента (подразумевая, что `sp` будет применяться к каждому изображению в наборе данных). Вызов `.load` в строке 29 загружает наш фактический набор данных изображения с диска. Этот метод возвращает два кортежа наших данных (каждое изображение изменено на 32×32 пикселя) вместе с метками для каждого изображения.

После загрузки наших изображений с диска массив данных NumPy имеет `.shape` (`3000, 32, 32, 3`), что указывает на то, что в наборе данных 3000 изображений, каждое размером 32×32 пикселя с 3 каналами.

Однако, чтобы применить алгоритм k-NN, нам нужно «сгладить» наши изображения из 3D- представления в единый список интенсивностей пикселей. Мы делаем это, строка 30 вызывает метод `.reshape` для массива данных NumPy, объединяя изображения $32 \times 32 \times 3$ в массив с формой $(3000, 3072)$. Фактические данные изображения совсем не изменились — изображения просто представлены в виде списка из 3000 записей, каждая из которых имеет 3072 размера ($32 \times 32 \times 3 = 3072$).

Чтобы продемонстрировать, сколько памяти требуется для хранения этих 3000 изображений в памяти, в строках 33 и 34 вычисляется количество байтов, потребляемых массивом, а затем преобразуется это число в мегабайты).

Далее, давайте создадим наши тренировочные и тестовые сплиты (Шаг № 2 в нашем конвейере):

```

36 # кодировать метки как целые числа 37 le = LabelEncoder() 38 labels
= le.fit_transform(labels)
39
40 # разбить данные на обучающие и тестовые сплиты, используя 75% от 41 #
данные для обучения и оставшиеся 25% для тестирования 42 (trainX, testX, trainY,
testY) = train_test_split(data, labels, test_size=0.25, random_state= 42)
43

```

Строки 37 и 38 преобразуют наши метки (представленные в виде строк) в целые числа, где у нас есть одно уникальное целое число для каждого класса. Это преобразование позволяет нам сопоставить класс кошки с целым числом 0, класс собаки с целым числом 1 и класс панды с целым числом 2. Многие алгоритмы машинного обучения предполагают, что метки классов закодированы как целые числа, поэтому важно, чтобы мы получили в привычка выполнять этот шаг.

Вычисление наших сплитов для обучения и тестирования обрабатывается функцией `train_test_split` в строках 42 и 43. Здесь мы разделяем наши данные и метки на два уникальных набора: 75% данных для обучения и 25% для тестирования.

Обычно переменная `X` используется для ссылки на набор данных, который содержит точки данных, которые мы будем использовать для обучения и тестирования, тогда как у ссылается на метки классов (подробнее об этом вы узнаете в главе 8, посвященной параметризованному обучению). Поэтому мы используем переменные `trainX` и `testX` для ссылки на примеры обучения и тестирования соответственно. Переменные `trainY` и `testY` — это наши метки обучения и тестирования. Вы встретите эти общие обозначения в этой книге, а также в других книгах, курсах и руководствах по машинному обучению, которые вы можете прочитать.

Наконец, мы можем создать наш классификатор k-NN и оценить его (шаги № 3 и № 4 на изображении). конвейер классификации):

```
45 # обучение и оценка классификатора k-NN на необработанных интенсивностях пикселей 46 print("[INFO] оценка классификатора
k-NN...")
47 model = KNeighborsClassifier(n_neighbors=args["neighbours"],
48 n_jobs=args["jobs"])
49 model.fit(trainX, trainY)
50 print(classification_report(testY, model.predict(testX), target_names=le.classes_))
51
```

Строки 47 и 48 инициализируют класс `KNeighborsClassifier`. Вызов метода `.fit` в строке 49 «обучает» классификатор, хотя здесь не происходит фактического «обучения» — модель k-NN просто хранит данные `trainX` и `trainY` внутри, чтобы она могла создавать прогнозы при тестировании. устанавливается путем вычисления расстояния между входными данными и данными `trainX`.

Строки 50 и 51 оценивают наш классификатор с помощью функции `classification_report`. Здесь нам нужно предоставить метки класса `testY`, предсказанные метки классов из нашей модели и, необязательно, имена меток классов (например, «собака», «кошка», «панда»).

7.2.4 Результаты k-NN

Чтобы запустить наш классификатор k-NN, выполните следующую команду:

```
$ python knn.py --dataset ../datasets/animals
```

Затем вы должны увидеть следующий вывод, похожий на следующий:

```
[INFO] загрузка изображений...
[INFO] обработано 500/3000
[INFO] обработано 1000/3000
[INFO] обработано 1500/3000
[INFO] обработано 2000/3000
[INFO] обработано 2500/3000
[INFO] обработано 3000/3000
[INFO] матрица характеристик: 9.0MB
[INFO] оценка классификатора k-NN...
```

	точность	вспомнить поддержку	f1-score	
кошки	0,39	0,49	0,43	239
	0,36	0,47	0,41	249
	0,79	0,36	0,50	262
среднее / общее	0,52	0,44	0,45	750

Обратите внимание, что наша матрица признаков потребляет всего 9 МБ памяти для 3000 изображений, каждое из которых имеет размер $32 \times 32 \times 3$ — этот набор данных без проблем можно хранить в памяти на современных машинах.

Оценивая наш классификатор, мы видим, что мы получили точность 52% — это неплохая точность для классификатора, который вообще не выполняет никакого настоящего «обучения», учитывая, что вероятность случайного угадывания правильный ответ 1/3.

Однако интересно проверить точность для каждой из меток класса. Класс "панда" был правильно классифицирован в 79% случаев, вероятно, из-за того, что панды в основном черные и белый, и, таким образом, эти изображения располагаются ближе друг к другу в нашем 3072-мерном пространстве.

Собаки и кошки получают значительно более низкую точность классификации — 39% и 36% соответственно. Эти результаты можно объяснить тем, что собаки и кошки могут иметь очень похожие оттенки меха.

Пальто и цвет их пальто не могут быть использованы, чтобы различить их. Фоновый шум (например, трава на заднем дворе, цвет дивана, на котором отдыхает животное и т. д.) также могут «запутать» алгоритм k-NN, поскольку он не может изучить какие-либо отличительные закономерности между этими видами. Этот путаница является одним из основных недостатков алгоритма k-NN: хотя он прост, он также не может учиться на данных.

В нашей следующей главе мы обсудим концепцию параметризованного обучения, где мы действительно можем изучать закономерности на самих изображениях, а не предполагайте, что изображения с похожим содержанием будут группируются в n-мерном пространстве.

7.2.5 Плюсы и минусы k-NN

Одним из основных преимуществ алгоритма k-NN является то, что его чрезвычайно просто реализовать и понять.

Кроме того, классификатор абсолютно не требует времени для обучения, так как все, что нам нужно сделать, это сохранить наши данные. точек с целью последующего вычисления расстояний до них и получения нашей окончательной классификации.

Однако мы платим за эту простоту во время классификации. Классификация новой контрольной точки требует сравнения с каждой отдельной точкой данных в наших обучающих данных, которые масштабируются $O(N)$, что делает работу с большими наборами данных непомерно вычислительна.

Мы можем бороться с этой стоимостью времени, используя алгоритмы приближенного ближайшего соседа (ANN). (например, kd-деревья [65], FLANN [66], случайные проекции [67, 68, 69] и т. д.); однако с помощью этих алгоритмов требуют, чтобы мы обменивали пространственно-временную сложность на «правильность» нашего ближайшего соседа алгоритм, так как мы выполняем приближение. Тем не менее, во многих случаях это того стоит. усилия и небольшие потери в точности для использования алгоритма k-NN. Такое поведение отличается от большинства алгоритмов машинного обучения (и все нейронные сети), на которые мы тратим большое количество времени предварительное обучение нашей модели для получения высокой точности и, в свою очередь, очень быстрой классификации в время тестирования.

Наконец, алгоритм k-NN больше подходит для низкоразмерных пространств признаков (которые изображения не). Расстояния в многомерных пространствах признаков часто не интуитивны, что вы можете прочитать подробнее об этом в превосходной статье Педро Доминго [70].

Также важно отметить, что алгоритм k-NN на самом деле ничему не «обучается». алгоритм не может сделать себя умнее, если ошибается; это просто зависит от расстояний в n-мерное пространство для классификации.

Учитывая эти минусы, зачем вообще изучать алгоритм k-NN? Причина в том, что алгоритм прост. Это легко понять. И самое главное, это дает нам базовый уровень, который мы

можно использовать для сравнения нейронных сетей и сверточных нейронных сетей по мере продвижения по остальной части этой книги.

7.3 Резюме

В этой главе мы узнали, как создать простой процессор изображений и загрузить набор данных изображения в память. Затем мы обсудили классификатор k-ближайших соседей, или сокращенно k-NN.

Алгоритм k-NN классифицирует неизвестные точки данных, сравнивая неизвестные точки данных с каждой точкой данных в обучающем наборе. Сравнение выполняется с использованием функции расстояния или метрики сходства. Затем из наиболее k похожих примеров в обучающей выборке мы накапливаем количество «голосов» за каждую метку. Категория с наибольшим количеством голосов «побеждает» и выбирается в качестве общей классификации.

Несмотря на простоту и интуитивность, алгоритм k-NN имеет ряд недостатков. Во-первых, он на самом деле ничего не «узнает» — если алгоритм делает ошибку, у него нет возможности «исправить» и «улучшить» себя для последующих классификаций. Во-вторых, без специализированных структур данных алгоритм k-NN масштабируется линейно с количеством точек данных, что делает его не только практически сложным для использования в больших размерностях, но и теоретически сомнительным с точки зрения его использования [70].

Теперь, когда мы получили основу для классификации изображений с использованием алгоритма k-NN, мы можем перейти к параметризованному обучению, фундаменту, на котором построены все глубокое обучение и нейронные сети. Используя параметризованное обучение, мы действительно можем учиться на наших входных данных и обнаруживать лежащие в основе закономерности. Этот процесс позволит нам создавать высокоточные классификаторы изображений, которые значительно увеличивают производительность k-NN.

8. Параметризованное обучение

В нашей предыдущей главе мы узнали о классификаторе k-NN — модели машинного обучения, настолько простой, что она вообще не выполняет никакого фактического «обучения». Нам просто нужно хранить данные обучения внутри модели, а затем во время тестирования делаются прогнозы путем сравнения точек данных тестирования с нашими данными обучения.

Мы уже обсудили многие плюсы и минусы k-NN, но в контексте крупномасштабных наборов данных и глубокого обучения самым запретительным аспектом k-NN являются сами данные. Хотя обучение может быть простым, тестирование выполняется довольно медленно, а узким местом является вычисление расстояния между векторами. Вычисление расстояний между точками обучения и тестирования масштабируется линейно с количеством точек в нашем наборе данных, что делает этот метод непрактичным, когда наши наборы данных становятся довольно большими. И хотя мы можем применять методы приближенного ближайшего соседа, такие как ANN [71], FLANN [66] или Annoy [72], для ускорения поиска, это все равно не решает проблему, заключающуюся в том, что k-NN не может функционировать без поддержания реплика данных внутри экземпляра (или, по крайней мере, иметь указатель на обучающий набор на диске и т. д.)

Чтобы понять, почему хранение точной копии обучающих данных внутри модели является проблемой, рассмотрите возможность обучения модели k-NN, а затем развертывания ее для клиентской базы из 100, 1000 или даже 1 000 000 пользователей. Если ваш тренировочный набор составляет всего несколько мегабайт, это может не быть проблемой, но если ваш тренировочный набор измеряется от гигабайт до терабайт (как в случае со многими наборами данных, к которым мы применяем глубокое обучение), у вас есть реальная проблема на ваши руки.

Рассмотрим обучающий набор набора данных ImageNet [42], который включает более 1,2 миллиона изображений. Если бы мы обучили модель k-NN на этом наборе данных, а затем попытались развернуть ее для набора пользователей, нам потребовалось бы, чтобы эти пользователи загрузили модель k-NN, которая внутренне представляет реплики 1,2 миллиона изображений. В зависимости от того, как вы скжимаете и храните данные, эта модель может измеряться от сотен гигабайт до терабайтов затрат на хранение и сетевых накладных расходов. Это не только пустая трата ресурсов, но и не оптимально для построения модели машинного обучения.

Вместо этого более желательным подходом было бы определение модели машинного обучения, которая может изучать шаблоны из наших входных данных во время обучения (что требует от нас тратить больше времени на процесс обучения), но имеет то преимущество, что определяется небольшим количеством параметры, которые можно легко использовать для представления модели, независимо от размера обучения. Этот тип машинного обучения называется

параметризованное обучение, которое определяется как:

«Модель обучения, которая суммирует данные с набором параметров фиксированного размера (независимого от количества обучающих примеров), называется параметрической моделью. Сколько бы данных вы не вводили в параметрическую модель, она не изменит своего мнения о том, сколько параметров ей нужно». - Рассел и Норвиг (2009) [73]

В этой главе мы рассмотрим концепцию параметризованного обучения и обсудим, как реализовать простой линейный классификатор. Как мы увидим далее в этой книге, параметризованное обучение является краеугольным камнем современного машинного обучения и алгоритмов глубокого обучения.



Большая часть этой главы была вдохновлена прекрасными заметками по линейной классификации Андрея Карпати в классе cs231n Стэнфорда [74]. Большое спасибо Karpathy и остальным ассистентам cs231n за составление таких доступных заметок.

8.1 Введение в линейную классификацию

Первая половина этой главы посвящена фундаментальной теории и математике, связанным с линейной классификацией, и в целом алгоритмам параметризованной классификации, которые изучают шаблоны из своих обучающих данных. Оттуда я предоставлю реальную реализацию линейной классификации и пример на Python, чтобы мы могли увидеть, как эти типы алгоритмов работают в коде.

8.1.1 Четыре компонента параметризованного обучения

Я уже несколько раз использовал слово «параметризованный», но что именно оно означает?

Проще говоря: параметризация — это процесс определения необходимых параметров данной модели. В задаче машинного обучения параметризация включает определение проблемы с точки зрения четырех ключевых компонентов: данных, функции оценки, функции потерь, а также весов и смещений.

Мы рассмотрим каждый из них ниже.

Данные

Этот компонент является нашими входными данными, из которых мы собираемся учиться. Эти данные включают в себя как точки данных (т. е. необработанные интенсивности пикселей изображений, извлеченные функции и т. д.), так и связанные с ними метки классов. Обычно мы обозначаем наши данные в терминах многомерной матрицы дизайна [10].

Каждая строка в матрице проекта представляет собой точку данных, в то время как каждый столбец (который сам может быть многомерным массивом) матрицы соответствует другому признаку. Например, рассмотрим набор данных из 100 изображений в цветовом пространстве RGB, каждое из которых имеет размер 32×32 пикселя.

Матрица дизайна для этого набора данных будет $X \in \mathbb{R}^{100 \times (32 \times 32 \times 3)}$, где X_i определяет i -е изображение в R . Используя это обозначение, X_1 — первое изображение, X_2 — второе изображение и так далее.

Наряду с матрицей дизайна мы также определяем вектор y , где y_i предоставляет метку класса для i -го примера в наборе данных.

Функция оценки

Функция оценки принимает наши данные в качестве входных данных и сопоставляет данные с метками классов. Например, учитывая наш набор входных изображений, функция оценки берет эти точки данных, применяет некоторую функцию f (наша функция оценки), а затем возвращает предсказанные метки классов, как в псевдокоде ниже:

```
INPUT_IMAGES => F(INPUT_IMAGES) => OUTPUT_CLASS_LABELS
```

Функция потери

Функция потерь количественно определяет, насколько хорошо наши предсказанные метки классов согласуются с нашими метками истинности.

Чем выше уровень согласия между этими двумя наборами меток, тем ниже наши потери (и выше точность нашей классификации, по крайней мере, на обучающем наборе).

Наша цель при обучении модели машинного обучения — минимизировать функцию потерь, тем самым повышая точность нашей классификации.

Веса и смещения Матрица

весов, обычно обозначаемая как W , и вектор смещения b называются весами или параметрами нашего классификатора, которые мы фактически будем оптимизировать. Основываясь на выводах нашей функции оценки и функции потерь, мы будем настраивать и возиться со значениями весов и смещений, чтобы повысить точность классификации.

В зависимости от типа вашей модели может существовать гораздо больше параметров, но на самом базовом уровне это четыре строительных блока параметризованного обучения, с которыми вы обычно сталкиваетесь. После того, как мы определили эти четыре ключевых компонента, мы можем применить методы оптимизации, которые позволят нам найти набор параметров W и b , которые минимизируют нашу функцию потерь по отношению к нашей функции оценки (при этом повышая точность классификации наших данных).

Далее, давайте посмотрим, как эти компоненты могут работать вместе, чтобы построить линейный классификатор, транс преобразование входных данных в фактические прогнозы.

8.1.2 Линейная классификация: от изображений к меткам

В этом разделе мы рассмотрим более математическую мотивацию подхода параметризованной модели к машинному обучению.

Для начала нам нужны наши данные. Предположим, что наш обучающий набор данных обозначен как x_i , где каждое изображение имеет связанную метку класса y_i . Предположим, что $i = 1, \dots, N$ и $y_i = 1, \dots, K$, подразумевая, что у нас есть N точек данных размерности D , разделенных на K уникальных категорий.

Чтобы сделать эту идею более конкретной, рассмотрим наш набор данных «Животные» из главы 7. В этом наборе данных всего $N = 3000$ изображений. Каждое изображение имеет размер 32×32 пикселя и представлено в цветовом пространстве RGB (т. е. три канала на изображение). Мы можем представить каждое изображение как $D = 32 \times 32 \times 3 = 3072$ различных значений. Наконец, мы знаем, что всего имеется $K = 3$ метки классов: по одной для классов собак, кошек и панд соответственно.

Учитывая эти переменные, теперь мы должны определить функцию оценки f , которая сопоставляет изображения с метки классов. Один из методов подсчета очков — это простое линейное отображение:

$$f(x_i, W, b) = Wx_i + b \quad (8.1)$$

Предположим, что каждый x_i представлен как один вектор-столбец с формой $[D \times 1]$ (в этом примере мы бы сгладили изображение $32 \times 32 \times 3$ в список из 3072 целых чисел). Тогда наша весовая матрица W будет иметь форму $[K \times D]$ (количество меток классов по размерности входных изображений). Наконец, b , вектор смещения будет иметь размер $[K \times 1]$. Вектор смещения позволяет нам сдвигать и переводить нашу оценочную функцию в том или ином направлении, фактически не влияя на нашу весовую матрицу W . Параметр смещения часто имеет решающее значение для успешного обучения.

Возвращаясь к примеру с набором данных Animals, каждый x_i представлен списком из 3072 значений пикселей, поэтому x_i имеет форму $[3072 \times 1]$. Весовая матрица W будет иметь форму $[3 \times 3,072]$, и, наконец, вектор смещения b будет иметь размер $[3 \times 1]$.

Рисунок 8.1 следует за иллюстрацией оценочной функции линейной классификации f . Слева у нас есть исходное входное изображение, представленное как изображение $32 \times 32 \times 3$. Затем мы объединяем это изображение в список из 3072 пикселей с интенсивностью, беря трехмерный массив и преобразуя его в одномерный список.

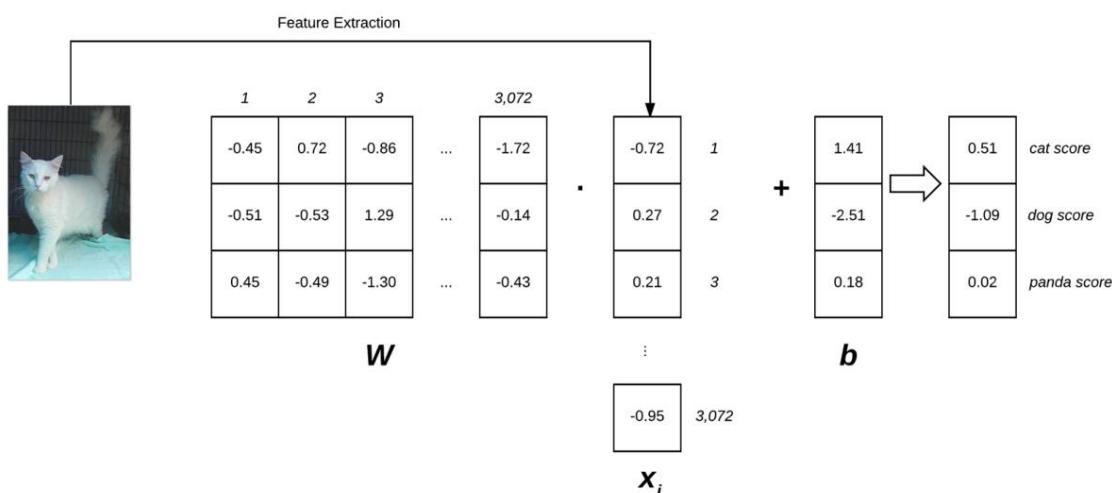


Рисунок 8.1: Иллюстрация скалярного произведения весовой матрицы W и вектора признаков x с последующим добавлением члена смещения. Рисунок вдохновлен примером Карпати из курса cs231n Стэнфордского университета [57].

Наша весовая матрица W содержит три строки (по одной для каждой метки класса) и 3072 столбца (по одному для каждого пикселя изображения). После скалярного произведения между W и x_i мы добавляем вектор смещения b и есть наша фактическая функция оценки. Наша функция оценки дает три значения справа: оценки, связанные с ярлыками собаки, кошки и панды соответственно.

Читатели, не знакомые с приемом точечных продуктов, должны прочитать это краткое и краткое руководство: <http://pyimg.co/fgcvp>. Читателям, заинтересованным в углубленном изучении линейной алгебры, я настоятельно рекомендую ознакомиться с книгой Филипа Кляйна «Кодирование матричной линейной алгебры в приложениях к информатике» [75].

Глядя на приведенный выше рисунок и уравнение, вы можете убедиться, что входные данные x_i и u_i фиксированы и не могут быть изменены. Конечно, мы можем получить разные оси, применяя различные преобразования к входному изображению, но как только мы передаем изображение в функцию оценки, эти значения не меняются. Фактически, единственными параметрами, которые мы можем контролировать (с точки зрения параметризованного обучения), являются наша весовая матрица W и наш вектор смещения b . Таким образом, наша цель состоит в том, чтобы использовать как нашу функцию оценки, так и функцию потерь для оптимизации (т. е. систематического изменения) векторов веса и смещения, чтобы повысить точность нашей классификации.

То, как именно мы оптимизируем матрицу весов, зависит от нашей функции потерь, но обычно включает некоторую форму градиентного спуска. Мы рассмотрим функции потерь позже в этой главе. Методы оптимизации, такие как градиентный спуск (и его варианты), будут обсуждаться в главе 9. Однако на данный момент просто поймите, что, учитывая функцию оценки, мы также определим функцию потерь, которая говорит нам, насколько «хороши» наши прогнозы на входные данные.

8.1.3 Преимущества параметризованного обучения и линейной классификации Использование

параметризованного обучения имеет два основных преимущества:

1. Как только мы закончим обучение нашей модели, мы можем отбросить входные данные и оставить только весовую матрицу W и вектор смещения b . Это существенно уменьшает размер нашей модели, поскольку нам нужно хранить два набора векторов (вместо всего обучающего набора).
2. Классификация новых тестовых данных выполняется быстро. Чтобы выполнить классификацию, все, что нам нужно сделать, это взять скалярное произведение W и x_i , а затем добавить смещение b (т. е. применить нашу

функция). Делать это таким образом значительно быстрее, чем сравнивать каждую тестовую точку с каждым обучающим примером, как в алгоритме k-NN.

8.1.4 Простой линейный классификатор с Python

Теперь, когда мы рассмотрели концепцию параметризованного обучения и линейной классификации, давайте реализуем очень простой линейный классификатор с помощью Python.

Цель этого примера не в том, чтобы продемонстрировать, как мы обучаем модель от начала до конца (мы рассмотрим это в одной из последующих глав, так как у нас еще есть кое-что, что нужно изучить, прежде чем мы будем готовы обучать модель с нуля). Но просто показать, как мы будем инициализировать весовую матрицу W , вектор смещения b , а затем использовать эти параметры для классификации изображения с помощью простого скалярного произведения.

Давайте продолжим и начнем этот пример. Наша цель здесь — написать скрипт Python, который будет правильно классифицировать рис. 8.2 как «собаку».



Рисунок 8.2: Наш пример входного изображения, которое мы собираемся классифицировать с помощью простого линейного классификатора.

Чтобы увидеть, как мы можем выполнить эту классификацию, откройте новый файл, назовите его `linear_example.py`, и вставьте следующий код:

```

1 # импортируем необходимые пакеты 2
импортируем numpy as np 3 импортируем
cv2
4
5 # инициализируем метки класса и устанавливаем начальное число генератора
псевдослучайных чисел 6 # чтобы мы могли воспроизвести наши результаты 7 labels =
["dog", "cat", "panda"] 8 np.random.seed(1)

```

Строки 2 и 3 импортируют необходимые пакеты Python. Мы будем использовать NumPy для наших числовых обработки и OpenCV для загрузки нашего примера изображения с диска.

Строка 7 инициализирует список меток целевого класса для набора данных «Животные», а строка 8 устанавливает генератор псевдослучайных чисел для NumPy, гарантируя, что мы сможем воспроизвести результаты этого эксперимента.

Далее давайте инициализируем нашу весовую матрицу и вектор смещения:

```

10 # случайным образом инициализируем нашу весовую матрицу и вектор
смещения -- в 11 # *реальной* задаче обучения и классификации эти параметры

```

```
12 # быть *узнанным* нашей моделью, но ради этого примера 13 # давайте использовать случайные значения
```

```
14 W = np.random.randn(3, 3072) 15 b =
np.random.randn(3)
```

Строка 14 инициализирует весовую матрицу W случайными значениями из равномерного распределения, выбранными в диапазоне $[0,1]$. Эта весовая матрица имеет 3 строки (по одной для каждой метки класса) и 3072 столбца (по одному для каждого пикселя в нашем изображении $32 \times 32 \times 3$).

Затем мы инициализируем вектор смещения в строке 15 — этот вектор также случайным образом заполняется значениями, равномерно отобранными по распределению $[0,1]$. Наш вектор смещения имеет 3 строки (соответствующие количеству меток классов) и один столбец.

Если бы мы обучали этот линейный классификатор с нуля, нам нужно было бы узнать значения W и b в процессе оптимизации. Однако, поскольку мы еще не достигли этапа оптимизации обучения модели, я инициализировал генератор псевдослучайных чисел значением 1, чтобы убедиться, что случайные значения дают нам «правильную» классификацию (я заранее протестировал случайные значения инициализации, чтобы определить, какие значение дает нам правильную классификацию). А пока просто относитесь к весовой матрице W и вектору смещения b как к «массивам черного ящика», оптимизированным волшебным образом — мы приоткроем завесу и покажем, как эти параметры изучаются в следующей главе.

Теперь, когда наша весовая матрица и вектор смещения инициализированы, давайте загрузим наш пример изображения с диска:

```
17 # загрузите наш пример изображения, измените его размер, а затем сгладьте его
в соответствии с нашим 18 # представлением "вектора признаков" 19 orig =
cv2.imread("beagle.png") 20 image = cv2.resize(orig, (32, 32)).сгладить()
```

Строка 19 загружает наше изображение с диска через `cv2.imread`. Затем мы изменяем размер изображения до 32×32 пикселей (игнорируя соотношение сторон) в строке 20 — наше изображение теперь представлено в виде $(32, 32, 3)$ массива NumPy, который мы сглаживаем до 3072-мерного вектора.

Следующим шагом является вычисление оценок меток выходного класса с помощью нашей функции оценки:

```
22 # вычислить выходные баллы, взяв скалярное произведение между 23 # весовой матрицей и пикселями изображения, с последующим
добавлением смещения 24 scores = W.dot(image) + b
```

Строка 24 — это сама функция оценки — это просто скалярное произведение между матрицей весов W и интенсивностью пикселей входного изображения, за которым следует добавление смещения b .

Наконец, наш последний блок кода обрабатывает запись значений функции оценки для каждого из классов. метки на наш терминал, затем отображая результат на нашем экране:

```
26 # перебираем оценки + метки и отображаем их 27 for (метка,
оценка) в формате zip(метки, оценки):
28     print("[INFO] {}: {:.2f}".format(label, score))
29
30 # нарисовать метку с наивысшим баллом на изображении в качестве
нашего прогноза 31 # 32 cv2.putText(orig, "Label:
{}".format(labels[np.argmax(scores)]), (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255,
33             0), 2)
34
```

```
35 # отображаем наше вх одное
изображение 36 cv2.imshow("Image",
orig) 37 cv2.waitKey(0)
```

Чтобы выполнить наш пример, просто введите следующую команду:

```
$ python linear_example.py
[ИНФОРМАЦИЯ] собака: 7963,93
[ИНФОРМАЦИЯ] кошка: -2930,99
[ИНФОРМАЦИЯ] панда: 3362,47
```

Обратите внимание, что класс собаки имеет наибольшее значение функции оценки, что означает, что класс «собака» будет выбран в качестве прогноза нашим классификатором. На самом деле мы можем видеть правильно нарисованную текстовую собаку на нашем вх одног изображении (рис. 8.2) на рис. 8.3.

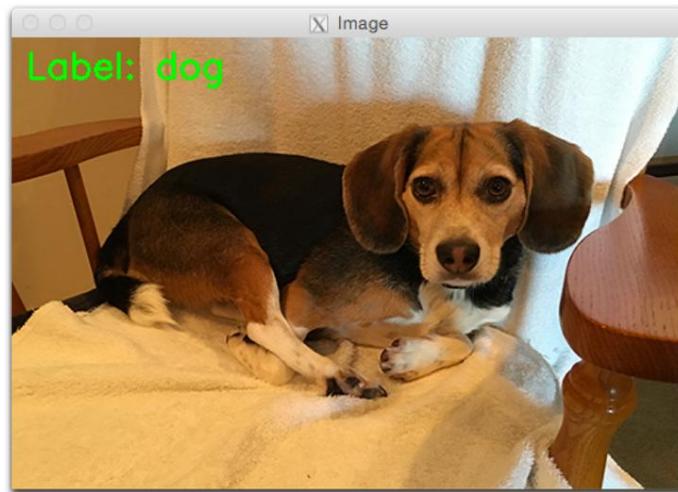


Рисунок 8.3: В этом примере нашлинейный классификатор смог правильно поставить вх одног изображение как собаку; однако имейте в виду, что этот рабочий пример. Далее в этой книге вы узнаете, как настроить наши веса и смещения, чтобы они автоматически делали эти прогнозы.

Опять же, имейте в виду, что это был рабочий пример. Я намеренно установил случайное состояние нашего скрипта Python для генерации значений W и b , которые привели бы к правильной классификации (выможете изменить псевдослучайное начальное значение в строке 8, чтобы увидеть, как разные случайные инициализации будут давать разные вх одног прогнозы).

На практике вы никогда не инициализируете свои значения W и b и не предполагаете, что они дадут вам правильную классификацию без какого-либо процесса обучения. Вместо этого при обучении наших собственных моделей машинного обучения с нуля нам нужно будет оптимизировать и изучить W и b с помощью алгоритма оптимизации, такого как градиентный спуск.

Мы рассмотрим оптимизацию и градиентный спуск в следующей главе, а пока просто найдите время, чтобы убедиться, что вы понимаете строку 24 и то, как линейный классификатор выполняет классификацию, взвесив скалярное произведение между матрицей весов и точкой вх одног . С последующим добавлением смещения. Таким образом, в свою модель можно определить с помощью двух значений: весовой матрицы вектора смещения. Это представление не только компактно, но и достаточно мощно, когда мы обучаем модели машинного обучения с нуля.

8.2 Роль функций потерь

В нашем последнем разделе мы обсудили концепцию параметризованного обучения. Этот тип обучения позволяет нам брать наборы входных данных и метки классов и фактически изучать функцию, которая отображает входные данные в выходные прогнозы, определяя набор параметров и оптимизируя их.

Но для того, чтобы на самом деле «вывучить» отображение входных данных на метки классов с помощью нашей функции оценки, нам нужно обсудить две важные концепции:

1. Функции потерь

2. Методы оптимизации.

Составляющая часть этой главы посвящена обычным функциям потерь, с которыми вы столкнетесь при построении нейронных сетей и сетей глубокого обучения. Глава 9 Starter Bundle полностью посвящена основным методам оптимизации, а глава 7 Practitioner Bundle посвящена более продвинутым методам оптимизации.

Отметь же, эта глава предназначена для краткого обзора функций потерь и их роли в параметризованном обучении. Подробное обсуждение функций потерь входит за рамки этой книги, и я очень рекомендую курс Эндрю Ng на Coursera [76], Witten et al. [77], Harrington [78] и Marsland [79], если вы хотите дополнить эту главу более математически строгими выводами.

8.2.1 Что такое функции потерь?



Рисунок 8.4: Потери при обучении для двух отдельных моделей, обученных на наборе данных CIFAR-10, нанесены во времени. Наша функция потерь количественно определяет, насколько «хороша» или «плоха» выполнена данная модель работу по классификации точек данных из набора данных. Модель № 1 обеспечивает значительно меньшие потери, чем модель № 2.

На самом базовом уровне функция потерь количественно определяет, насколько «хорошим» или «плохим» является данный предиктор при классификации точек входных данных в наборе данных. Визуализация функций потерь, построенных со временем.

для двух отдельных моделей, обученных на наборе данных CIFAR-10, показано на рис. 8.4. Чем меньше потери, тем лучше классификатор справляется с моделированием взаимосвязи между векторами данными и метками векторов одних классов (х отя есть момент, когда мы можем переобучить нашу модель — при слишком точном моделировании обучавших данных наша модель теряет способность общаться, явление, которое мы подробно обсудим в главе 17). И наоборот, чем больше наша потеря тем больше работы необходимо проделать для повышения точности классификации.

Чтобы повысить точность нашей классификации, нам нужно настроить параметры нашей весовой матрицы W или вектора смещения b . То, как именно мы будем эти параметры, является проблемой оптимизации, которую мы рассмотрим в следующей главе. А пока просто поймите, что функцию потерь можно использовать для количественной оценки того, насколько хорошо наша оценочная функция справляется с классификацией точек в векторах данных.

В идеале наши потери должны уменьшаться со временем, когда мы настраиваем параметры нашей модели. Как показано на рис. 8.4, потери модели № 1 начинутся немного выше, чему модели № 2, но затем быстро уменьшаются и продолжают оставаться низкими при обучении на наборе данных CIFAR-10. И наоборот, потери для модели № 2 сначала уменьшаются, но быстро стагнируют. В этом конкретном примере модель № 1 обеспечивает меньшие общие потери и, вероятно, является более желательной моделью для использования при классификации других изображений из набора данных CIFAR-10. Я говорю «вероятно», потому что есть шанс, что модель № 1 переобучится данным обучения. Мы рассмотрим эту концепцию переобучения и способность обнаружения в главе 17.

8.2.2 Потеря мультиклассового SVM

Multi-class SVM Loss (как следует из названия) вдохновлен линейными машинами опорных векторов (SVM) [43], которые используют функцию оценки f для сопоставления наших точек данных с числовыми оценками для каждой метки класса. Эта функция f представляет собой простое обучение отображения:

$$f(x_i, W, b) = Wx_i + b \quad (8.2)$$

Теперь, когда у нас есть функция оценки, нам нужно определить, насколько «хорошей» или «плохой» является эта функция (учитывая матрицу весов W и вектор смещения b) при прогнозировании. Чтобы сделать это определение, нам нужна функция потерь.

Напомним, что при создании модели машинного обучения у нас есть матрица дизайна X , где каждая строка в X содержит точку данных, которую мы хотим классифицировать. В контексте классификации изображений каждая строка в X является изображением, и мы стремимся правильно пометить это изображение. Мы можем получить доступ к i -му изображению внутри X через синтаксис x_i .

Точность также у нас также есть вектор y , который содержит наши метки классов для каждого x . Эти значения являются нашими метками истинности, и мы надеемся, что наша функция оценки будет правильно предсказывать. Точность же, как мы можем получить доступ к данному изображению как x_i , мы можем получить доступ к связанной метке класса через y_i .

Для простоты давайте сократим нашу функцию оценки как s :

$$s = f(x_i, W) \quad (8.3)$$

Это означает, что мы можем получить прогнозируемый балл j -го класса через i -ю точку данных:

$$s_j = f(x_i, W)_j \quad (8.4)$$

Используя этот синтаксис, мы можем собрать все вместе, получив функцию потери шарнира:

$$\text{Ли} = \max_{j \neq y_i} (0, s_j - s_{y_i} + 1) \quad (8.5)$$

 Почти все функции потерь включают член регуляризации. Я пока опускаю эту идею, так как мы рассмотрим регуляризацию в главе 9, когда лучше поймем функции потерь.

Глядя на приведенное выше уравнение потерь на шарнире, вы можете быть сбиты с толку тем, что оно на самом деле делает. Последний, функция потери шарнира суммирует все неправильные классы ($i = j$) и сравнивает в них одни данные нашей оценочной функции s , возвращенные для метки j -го класса (неправильный класс) и y_i -го класса (правильный класс). Мы применяем операцию \max , чтобы зафиксировать значения на нуле, что важно, чтобы гарантировать, что мы не суммируем отрицательные значения.

Данное χ_i классифицируется правильно, когда потери $L_i = 0$ (я приведу численный пример в следующем разделе). Чтобы получить потери по всему нашему тренировочному набору, мы просто берем среднее значение для каждого отдельного L_i :

$$L = \frac{1}{N} \sum_{i=1}^N L_i \quad (8.6)$$

Другая связанныя функция потерь, с которой вы можете столкнуться — это квадратная потеря шарнира:

$$L_i = \max(0, s_j - s_{y_i} + 1)^2 \quad (8.7)$$

Квадрат члена более сильно наказывает наши потери, введя в квадрат в выпуске, что приводит к квадратичный рост потерь в неверном прогнозе (по сравнению с линейным ростом).

Что касается того, какую функцию потерь вы должны использовать, это полностью зависит от вашего набора данных. Типично видеть, что стандартная функция потери шарнира используется чаще, но в некоторых наборах данных квадратичная вариация может иметь лучшую точность. В целом, это гиперпараметр, который следует настроить.

Пример потери SVM для нескольких

классов Теперь, когда мы рассмотрели математику потери шарнира, давайте рассмотрим рабочий пример. Мы снова будем использовать набор данных «Животные», цель которого — классифицировать данное изображение как содержащее кошку, собаку или панду. Для начала взгляните на рисунок 8.5, где я включил три обучения из трех классов набора данных «Животные».

При заданной произвольной весовой матрице W и векторе смещения в них одни оценки $f(x, W) = Wx + b$ отображаются в теле матрицы. Чем больше баллов, тем увереннее наша функция оценки в отношении прогноза.

Начнем вычисления потерь L_i для класса «собака»:

1 >>> max(0, 1.33 - 4.26 + 1) + max(0, -1.01 - 4.26 + 1) 2 0

Обратите внимание, что выше уравнение здесь включает два члена — разницу между прогнозируемой оценкой собаки и оценкой кошки и панды. Также обратите внимание на то, что потеря для «собаки» равна нулю — это означает, что собака была предсказана правильно. Быстрое изучение изображения № 1 на рис. 8.5 выше показывает, что этот результат верен: оценка «собаки» больше, чем оценки «кошки» и «панды».

Точно так же мы можем вычислить потери на петлях для изображения № 2, на котором изображена кошка:

3 >>> max(0, 3.76 - (-1.20) + 1) + max(0, -3.81 - (-1.20) + 1)
4 5.96



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

Рисунок 8.5: В верхней части рисунка у нас есть три вх однък изображения по одному для каждого класса собак, кошек и панд соответственно. Основная часть таблицы содержит вх однъе данные функции оценки для каждого из классов. Мы будем использовать функцию подсчета очков, чтобы получить общую потерю для каждого вх однъого изображения.

В этом случае наша функция потерь больше нуля что указывает на то, что наш прогноз неверен. Глядя на нашу функцию оценки, мы видим, что наша модель предсказывает собаку в качестве предложенного ярлыка с оценкой 3,76 (поскольку это ярлык с наивысшим баллом). Мы знаем, что эта метка неверна, и в главе 9 мы узнаем, как автоматически настраивать наши веса, чтобы исправить эти предсказания.

Наконец, давайте вычислим потерю на петлях для примера с пандой:

```
5 >>> max(0, -2.37 - (-2.27) + 1) + max(0, 1.03 - (-2.27) + 1)
6 5.199999999999999
```

Слить же, наша потеря равна нулю, поэтому мы знаем, что у нас неверный прогноз. Глядя на нашу оценку функция наша модель неправильно обозначила это изображение как «кошка», хотя оно должно было быть «пандой».

Затем мы можем получить общие потери по трем примерам, взяв среднее значение:

```
7 >>> (0,0 + 5.96 + 5,2) / 3,0
8 3,72
```

Таким образом, учитывая наши три тренировочных примера, наша общая потеря шарнира составляет 3,72 для параметров W и b.

Также обратите внимание, что наша потеря была нулевой только для одного из трех вх однък изображений, что означает, что два наших прогноза были неверны. В нашей следующей главе мы узнаем, как оптимизировать W и b, чтобы делать более точные прогнозы используя функцию потерь, которая поможет вести и направлять нас в правильном направлении.

8.2.3 Кросс-энтропийные потери и классификаторы Softmax

Хотя потеря шарнира довольно популярна, вы скорее всего, столкнетесь с кросс-энтропийной потерей и классификаторами Softmax в контексте глубокого обучения сверточных нейронных сетей.

Почему это? Проще говоря

классификаторы Softmax дают вам вероятности для каждой метки класса, а потеря шарнира дает вам запас.

Нам, людям, намного проще интерпретировать вероятности, а не оценки маржи. Кроме того, для наборов данных, таких как ImageNet, мы часто смотрим на точность ранга 5 сверточных нейронных сетей (где мы проверяем, входит ли метка истинности в топ-5 предсказанных меток, возвращаемая сетью для заданного в однозначения). изображение). Увидеть, существует ли (1) истинная метка класса в топ-5 предсказаний и (2) вероятность, связанная с каждой меткой, является ярким свидетельством.

Понимание перекрестной энтропийной потери

Классификатор Softmax представляет собой обобщение бинарной формул логистической регрессии. Точность также, как в случае потери шарнира или квадрата потери шарнира, наша функция отображения определена таким образом, что она принимает в один набор данных x_i и отображает их в метки в виде одного класса посредством скалярного произведения данных x_i и весовой матрицы W (опуская член смещения для краткости):

$$f(x_i, W) = Wx_i \quad (8.8)$$

Однако, в отличие от потери шарнира, мы можем интерпретировать эти оценки как ненормализованые логарифмические вероятности для каждой метки класса, что равносильно замене функции потери шарнира кросс-энтропийной потерь:

$$L_i = -\log(e^{s_{y_i}} / \sum_j e^{s_j}) \quad (8.9)$$

Итак, как ясно из формулы, мы разбираем функцию на части и посмотрим. Для начала наша функция потерь должна минимизировать отрицательную логарифмическую вероятность правильного класса:

$$L_i = -\log P(Y = y_i | X = x_i) \quad (8.10)$$

Утверждение вероятности можно интерпретировать как:

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad (8.11)$$

Где мы используем нашу стандартную форму функции оценки:

$$s = f(x_i, W) \quad (8.12)$$

В целом это дает нашу окончательную функцию потерь для одной точки данных, как выше:

$$L_i = -\log(e^{s_{y_i}} / \sum_j e^{s_j}) \quad (8.13)$$

Обратите внимание, что ваш логарифм здесь на самом деле является основанием (натуральным логарифмом), так как мы берем обратное введение в степень по сравнению. Фактическое введение в степень и нормализация через сумму показателей — это наша функция Softmax. Отрицательный журнал дает нашу фактическую потерю перекрестной энтропии.

Так же, как и в случае потери шарнира и квадрата потери шарнира, вычисление перекрестной энтропийной потери по всему набору данных выполняется путем взятия среднего значения

$$\Lambda = \frac{1}{N} \sum_{i=1}^N L_i \quad (8.14)$$

Следующим шагом в регуляризации из нашей функции потерь. Мы вернемся к регуляризации, объясним, что это такое, как ее использовать и почему она важна для нейронных сетей и глубокого обучения в главе 9. Если вы не знаете, что это такое, мы обсудим это в главе 9.

Числовые примеры в следующем разделе, чтобы убедиться, что вы понимаете, как работает кросс-энтропийная потеря

Рабочий пример Softmax

Scoring Function	
Dog	-3.44
Cat	1.16
Panda	3.91



Input Image

	Scoring Function	Unnormalized Probabilities
Dog	-3.44	0.03
Cat	1.16	3.19
Panda	3.91	49.90

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities
Dog	-3.44	0.0321	0.0006
Cat	1.16	3.1899	0.0601
Panda	3.91	49.8990	0.9393

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities	Negative Log Loss
Dog	-3.44	0.0321	0.0006	
Cat	1.16	3.1899	0.0601	
Panda	3.91	49.8990	0.9393	0.0626

Рисунок 8.6: Первая таблица: чтобы вычислить наши потери кросс-энтропии, давайте начнем с ввода функции оценки. Вторая таблица: возведение в степень в e одних значений функции оценки дает нам наши ненормализованные вероятности. Третья таблица: чтобы получить фактические вероятности, мы делим каждую отдельную ненормализованную вероятность на сумму всех ненормализованных вероятностей. Четвертая таблица: отрицательный натуральный логарифм вероятности правильной наземной истины дает окончательную потерю для точек данных.

Чтобы продемонстрировать кросс-энтропийную потерю в действии, рассмотрим рис. 8.6. Наша цель — классифицировать, содержит ли изображение выше собаку, кошку или панду. Ясно, мы видим, что изображение является «пандой», но что думает наш классификатор Softmax? Чтобы выяснить это, нам нужно изучить каждую из четырех таблиц на рисунке.

Первая таблица включает в e одни данные нашей функции оценки f для каждого из трех классов соответственно. Эти значения являются нашими ненормализованными логарифмическими вероятностями для трех классов. Возведем в степень результат оценочной функции ($f(x)$ называется ненормализованной вероятностью и обозначается p) и получим нормализованную вероятность (нормализованная вероятность p обозначается \hat{p}). Вторая таблица показывает, как это работает.

Следующим шагом является взятие знаменателя суммирования показателей и деление на сумму, что дает фактические вероятности, связанные с каждой меткой класса (третья таблица). Обратите внимание, что суммы вероятностей равны единице. Наконец, мы можем взять отрицательный натуральный логарифм, $-\ln(p)$, где p — нормализованная вероятность, что дает наш окончательный проигрыш (четвертая таблица).

В этом случае наш классификатор Softmax правилью сообщит об изображении как о панде с достоверностью 93,93%. Затем мы можем повторить этот процесс для всех изображений в нашем обучальном наборе, взять среднее значение и получить общую потерю перекрестной энтропии для обучения шаблона. Этот процесс позволяет нам качественно оценить, насколько хорошо или плохо работает набор параметров в нашем обучальном наборе.



Я использовал генератор случайных чисел, чтобы получить значения функции оценки для этого конкретного примера. Эти значения просто используются для демонстрации того, как включаются вычисления функции потери классификатора/перекрестной энтропии Softmax. На самом деле эти значения не будут генерироваться случайным образом — вместо этого они будут результатом нашей функции оценки f на основе ваших параметров W и b . Мы видим, как все компоненты параметризованного обучения сочетаются друг с другом в нашей следующей главе, но пока мы работаем с примерами, чтобы продемонстрировать, как работают функции потерь.

8.3 Резюме

В этой главе мы рассмотрели четыре компонента параметризованного обучения:

1. Данные
2. Функция подсчета ошибок.
3. Функция потерь.
4. Веса и смещения

в контексте классификации изображений наши вх однъе данные — это наш набор данных изображений. Функция подсчета ошибок производит прогнозы для каждого изображения. Затем функция потерь количественно определяет, насколько хороши или плохи имеются набор прогнозов для набора данных. Наконец, векторная матрица и вектор смещения — это то, что позволяет нам фактически «учиться» на вх однъе данных — эти параметры будут изменены настроены с помощью методов оптимизации в попытке добиться более высокой точности классификации.

Затем мы рассмотрели две популярные функции потерь: потери шарнира и потери перекрестной энтропии. Хотя потеря шарнира используется во многих приложениях машинного обучения (таких как SVM), я могу почти с абсолютной уверенностью гарантировать, что вы увидите потерю кросс-энтропии с самой частотой, в основном из-за того, что классификаторы Softmax в большинстве вероятности, а не поля. Нам, людям, гораздо легче интерпретировать вероятности, поэтому этот факт является особенно приятным качеством кросс-энтропийной потери и классификаторов Softmax. Для получения дополнительной информации о потерях шарнира потерь и кросс-энтропийной потерь, пожалуйста, обратитесь к курсу cs231n Стенфордского университета [57, 74].

В нашей следующей главе мы рассмотрим методы оптимизации, которые используются для настройки нашей векторной матрицы и вектора смещения. Методы оптимизации позволяют нашим алгоритмам фактически учиться на наших вх однъе данных, обновляя матрицу весов и вектор смещения на основе вх однъе данных наших функций оценки и потерь. Используя эти методы, мы можем постепенно приближаться к значениям параметров, обеспечивающим минимальные потери и более высокую точность. Методы оптимизации являются краеугольным камнем современных нейронных сетей и глубокого обучения и без них мы не смогли бы изучать шаблоны из наших вх однъе данных, поэтому обязательно обратите внимание на следующую главу.

9. Методы оптимизации и регуляризация

«Почти все глубокое обучение основано на одном очень важном алгоритме: Stochastic Gradient Descent (SGD)» — Goodfellow et al. [10]

На данный момент у нас есть четкое понимание концепции параметризованного обучения. В последних нескольких главах мы обсуждали концепцию параметризованного обучения и то, как этот тип обучения позволяет нам определить функцию оценки, которая сопоставляет наши вх однъе данные с метками вх однък классов.

Эта функция оценки определяется двумя важными параметрами; в частности, наша весовая матрица W и наш вектор смещения b . Наша функция оценки принимает эти параметры в качестве вх однък данных и возвращает прогноз для каждой точки вх однък данных x_i .

Мы также обсудили две общие функции потерь: потери мультиклассового SVM и кросс-энтропийные потери. Функции потерь на самом базовом уровне не используются для количественной оценки того, насколько «хорошим» или «плохим» является данный предиктор (т. е. набор параметров) при классификации точек вх однък данных в наших данных.

Учитывая эти строительные блоки, теперь мы можем перейти к наиболее важному аспекту машинного обучения нейронных сетей и глубокого обучения — оптимизации. Алгоритм оптимизации — это механиズмы, которые питают нейронные сети и позволяют им изучать закономерности на основе данных. В ходе этого обсуждения мы узнали, что получение высокоточного классификатора зависит от наших ожиданий набора весов W и b , чтобы наши точки данных были правильно классифицированы.

Но как нам найти и получить весовую матрицу W и вектор смещения b , обеспечивающие высокую точность классификации? Инициализируем ли наших случайным образом, оцениваем и повторяем снова и снова, надеясь, что в какой-то момент мы приведем наш набор параметров, который получит разумную классификацию? Мы могли бы это, учитывая, что современные сети глубокого обучения имеют параметры исчисляемые десятками миллионов, нам может потребоваться много времени, чтобы слепо наткнуться на разумный набор параметров.

Вместо этого, чтобы полагаться на чистую случайность, нам нужно определить алгоритм оптимизации, который позволит нам буквально улучшить W и b . В этой главе мы рассмотрим наиболее распространенный алгоритм, используемый для обучения нейронных сетей и моделей глубокого обучения — градиентный спуск. Градиентный спуск имеет много вариантов (которые мы также коснемся), но в каждом случае идея одна и та же: итеративно оценивайте свои параметры вниз и слейте свои потери, а затем делайте небольшой шаг в направлении, которое минимизирует ваши потери.

9.1 Градиентный спуск

Алгоритм градиентного спуска имеет две основные разновидности:

1. Стандартная «ванильная» реализация
2. Оптимизированная «стochastic» версия, которая использует сэмплинг.

В этом разделе мы рассмотрим базовую реализацию vanilla, чтобы сформировать основу для нашего понимания. После того, как мы разберемся основами градиентного спуска, мы перейдем к stochastic версии. Затем мы рассмотрим некоторые «навороты» вроде свистки, которые мы можем добавить к градиентному спуску, включая импульс и ускорение Нестерова.

9.1.1 Ландшафт потерь и поверхность оптимизации

Метод градиентного спуска представляет собой итеративный алгоритм оптимизации, который работает с ландшафтом потерь (также называемым поверхностью оптимизации). Пример канонического градиентного спуска состоит в том, чтобы визуализировать наши веса по оси x , а затем потери для данного набора весов по оси y (рис. 9.1, слева):

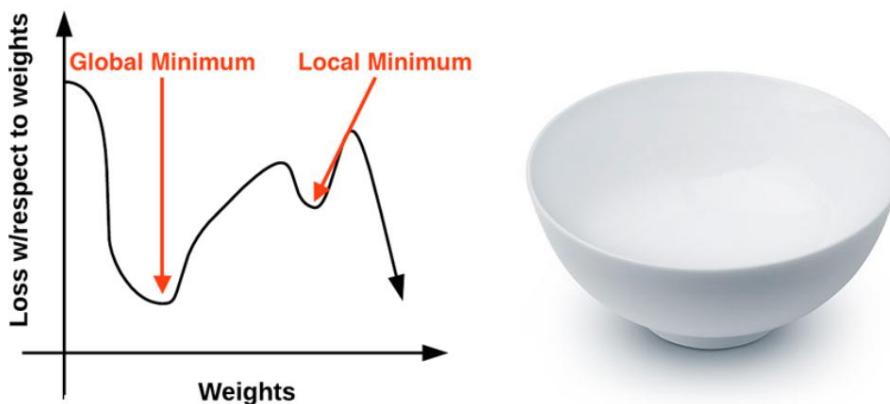


Рисунок 9.1: Слева: «Наша потеря», визуализированная в виде 2D-графика. Справа: более реалистичный ландшафт потерь можно представить в виде чаши, существующей в нескольких измерениях. Наша цель — применить градиентный спуск, чтобы перейти на дно этой чаши (где потери минимальны).

Как мы видим, наш ландшафт потерь имеет множество пиков и спадов, в зависимости от того, какие значения принимают наши параметры. Каждый пик представляет собой локальный максимум, представляющий областей с очень высокими потерями: локальный максимум с наибольшими потерями в нашем ландшафте потерь является глобальным максимумом. Точнее так же есть локальный минимум, который представляет собой множество небольших областей потерь.

Локальный минимум с наименьшими потерями в ландшафте потерь является нашим глобальным минимумом. В идеальном мире мы должны были бы найти этот глобальный минимум, чтобы наши параметры принимали наиболее оптимальные возможные значения.

Поэтому возникает вопрос: «Если мы достигли глобального минимума, почему было просто не перейти к нему напрямую? Это орошено посюжету?»

В этом проблема — ландшафт потерь для нас невидим. На самом деле мы не знаем, как это выглядит. Если бы мы были оптимизационным алгоритмом, то мы бы в слепую размещались где-то на графике, не имея представления о том, как выглядит ландшафт перед нами, и нам пришлось бы прокладывать свой путь к минимуму потерь, забравшись случайно на вершину локального максимума.

Лично мне никогда не нравилась эта визуализация ландшафта потерь — она слишком проста и часто приводит читателей к мысли, что градиентный спуск (и его варианты) в конечном итоге найдет либо локальный, либо глобальный минимум. Это утверждение неверно, особенно для сложных задач, и я об этом.

далее в этой главе. Вместо этого давайте посмотрим на другую визуализацию ландшафта потерь, которая как мне кажется лучше описывает проблему. Здесь у нас есть миска, похожая на ту, из которой можно есть хлопья или суп (рис. 9.1, справа).

Поверхность нашей чаши — это ландшафт потерь, представляющий собой график функции потерь. Разница между нашим ландшафтом потерь и вашей миской с хлопьями заключается в том, что наша миска с хлопьями существует только в трех измерениях, в то время как наш ландшафт потерь существует в многих измерениях, возможно, в десятках, сотнях или тысячах измерений.

Каждое положение на поверхности чаши соответствует определенному значению потерь при заданном наборе параметров W (весовая матрица) и b (вектор смещения). Наша цель — попробовать разные значения W и b , оценить их потери, а затем сделать шаг к более оптимальным значениям, которые (в идеале) имеют меньшие потери.

9.1.2 «Градиент» в градиентном спуске

Чтобы сделать наше объяснение градиентного спуска более интуитивным, давайте представим, что у нас есть робот — назовем его Чад (рис. 9.2, слева). Выполнив градиентный спуск, мы случайным образом бросаем Чада куда-нибудь в наш ландшафт потерь (рис. 9.2, справа).



Рисунок 9.2: Слева: наш робот, Чад. Справа: работа Чада — ориентироваться в нашем ландшафте потерь и спускаться на дно бассейна. К сожалению, единственный датчик, который Чад может использовать для управления своей навигацией, — это специальная функция называемая функцией потерь, L . Эта функция должна направлять его в область с меньшими потерями.

Теперь задача Чада — добраться до дна бассейна (где потери минимальны). Кажется достаточно просто, верно? Все, что нужно сделать Чаду, это ориентироваться так, чтобы он смотрел «вниз по склону», и ехать по склону, пока не достигнет дна чаши.

Но вот проблема: Чад не очень умный робот. У Чада есть только один датчик — этот датчик позволяет ему взять свои параметры W и b , а затем вычислить функцию потерь L . Таким образом, Чад может вычислить свое относительное положение на ландшафте потерь, но он совершенно не представляет, в каком направлении он движется. следуя сделать шаг, чтобы приблизиться к дну тазика.

Что делать Чаду? Ответ заключается в применении градиентного спуска. Все, что нужно сделать Чаду, — это следовать наклону градиента W . Мы можем вычислить градиент W во всем измерении, используя следующее уравнение:

$$\frac{d\phi(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (9.1)$$

В измерениях > 1 наш градиент становится вектором частных производных. Проблема с этим уравнением заключается в том, что

1. Это приближение к градиенту.
2. Это умчительно медленно.

На практике в место этого мы используем аналитический градиент. Метод точени быстр, но чрезвычайно сложен в реализации из-за частных производных и исчисления с несколькими переменными. Полный ввод многомерного исчисления используемого для обоснования градиентного спуска, включает за рамки этой книги. Если вам интересно узнать больше о числовых и аналитических градиентах, я бы посоветовал эту лекцию Зибулевского [80], заметки Эндрю Нг по машинному обучению cs229 [81], а также заметки cs231n [82].

Ради этого обсуждения просто усвойте, что такое градиентный спуск: попытка минимизировать наши параметры для низких потерь в высокой точности классификации с помощью итеративного процесса, предпринимающего шаги в направлении, которое минимизирует потери.

9.1.3 Рассматривайте это как выпуклую задачу (даже если это не так)

Использование чаши на рис. 9.1 (справа) в качестве визуализации ландшафта убытков также позволяет нам сделать важный вывод о современных нейронных сетях — мы рассматриваем ландшафт убытков как выпуклую проблему, даже если это не так. Если некоторая функция F выпукла, то все локальные минимумы также являются глобальными минимумами. Эта идея прекрасно подходит для визуализации чаши. Наш алгоритм оптимизации просто должен пристегнуть пару лыж в верхней части чаши, а затем медленно спускаться по склону, пока не достигнем дна.

Проблема в том, что почти все проблемы к которым мы применяем нейронные сети и алгоритмы глубокого обучения, не являются аккуратными выпуклыми функциями. Вместо этого внутри этой чаши мы найдем широкие долины, более поглощенные на каньоны крутые обрывы даже если, где потери резко падают только для того, чтобы снова резко подняться.

Учитывая невыпуклую природу наших наборов данных, почему мы применяем градиентный спуск? Ответ просто: потому что он делает достаточно хорошую работу. Цитируя Goodfellow et al. [10]:

«[Алгоритм] оптимизации может не гарантировать достижение даже локального минимума за разумное время, но он часто находит очень низкое значение функции [потери] достаточно быстро, чтобы быть полезным».

Мы можем ожидать нахождения локального/глобального минимума при обучении сети глубокого обучения, но это ожидание редко сбывается с реальностью. Вместо этого мы в конечном итоге находим область с низкими потерями — эта область может даже не быть локальным минимумом, но на практике оказывается, что этого достаточно.

9.1.4 Уловка предвзятости

Прежде чем мы перейдем к реализации градиентного спуска, я хочу уделить время обсуждению метода, называемого «трюком смещения», метода объединения нашей весовой матрицы W и вектора смещения в один параметр. Напомним из наших предыдущих решений, что наша функция оценки определяется как:

$$f(x_i, W, b) = Wx_i + b \quad (9.2)$$

Часто бывает утомительно отслеживать две отдельные переменные, как с точки зрения объяснения так и с точки зрения реализации — чтобы полностью избежать этой ситуации, мы можем объединить W и b вместе. Чтобы объединить как матрицу смещения так и матрицу весов, мы добавляем дополнительное измерение (т. е. столбец) к нашим в однм данным X , которое содержит константу 1 — это наше измерение смещения.

Обычно мы либо добавляем новое измерение к каждому отдельному x_i как первое измерение, либо как последнее измерение. На самом деле это имеет значения. Мы можем выбрать любое произвольное место для вставки

столбец единиц в нашу матрицу дизайна, пока он существует. Это позволяет нам переписать нашу функцию подсчета очков с помощью умножения одной матрицы

$$f(x_i, W) = Wx_i \quad (9.3)$$

Однако же, нам разрешено опускать здесь члены, так как он встроен в нашу весовую матрицу. В контексте наших предыдущих примеров в наборе данных «Животные» мы работали с изображениями $32 \times 32 \times 3$ с общим размером 3072 пикселя. Каждый x_i представлен вектором размером $[3072 \times 1]$. Добавление измерения с постоянным значением единицы теперь расширяет вектор до $[3073 \times 1]$. Точнее так же объединение матрицы смещения в веса также расширяет нашу матрицу весов W до $[3 \times 3073]$, а не $[3 \times 3072]$. Таким образом, мы можем рассматривать смещение как обучаемый параметр в матрице весов, который нам не нужно отслеживать в отдельной переменной.

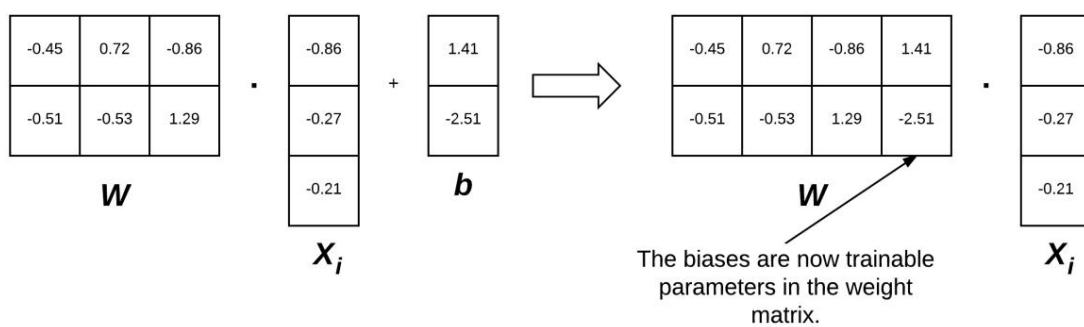


Рисунок 9.3: Слева: Обычно мы рассматриваем матрицу весов и вектор смещения как два отдельных параметра. Справа: однако мы можем встроить вектор смещения в матрицу весов (тем самым сделав его обучаемым параметром непосредственно внутри матрицы весов, инициализировав нашу матрицу весов дополнительным столбцом единиц).

Чтобы наглядно представить трюк с смещением, рассмотрите рисунок 9.3 (слева), где мы разделяем матрицу весов и смещение. До сих пор этот рисунок изображал то, как мы думали о нашей функции подсчета очков. Но вместо этого мы можем объединить W и b вместе, при условии, что мыставим новый столбец в каждый x_i , где каждая запись равна единице (рис. 9.3, справа). Применение трюка смещения позволяет нам изучить только одну матрицу весов, поэтому мы предпочитаем этот метод для реализации. Для всех будущих примеров в этой книге всякий раз, когда я буду упоминать W , предполагайте, что вектор смещения b также явно включен в матрицу весов.

9.1.5 Псевдокод для градиентного спуска

Ниже я включил Python-подобный псевдокод для стандартного алгоритма ванильного градиентного спуска (псевдокод, вдохновленный слайдами cs231n [83]):

```

1, x отя в ерно
2   Wградиент = оценка_градиента(потери, данные, В т)
3   W += -альфа * Wградиент

```

На этом псевдокоде построены все варианты градиентного спуска. Мы начинаем со строки 1, зацикливаясь до тех пор, пока не будет выполнено какое-либо условие, обычно:

- Прошло определенное количество эпох (это означает, что наш алгоритм обучения «увидел» каждую точку обучающих данных N раз).

2. Наши потери стали достаточно низкими или точность обучения удавлетворительно в высокой.

3. Потери не улучшились в последующих эпохах.

Затем в строке 2 вызывается функция с именем `Assessment_gradient`. Для этой функции требуются три параметра:

1. `loss`: функция используется для вычисления потерь по нашим текущим параметрам W и в однм днньем.

данные: наши обучавшие данные, где каждый обучавший образец представлен изображением (или функцией вектора).

3. W : наша фактическая матрица веса, которую мы оптимизируем. Наша цель — применить градиентный спуск, чтобы найти W , который дает минимальные потери.

Функция `Assessment_gradient` возвращает вектор, который является K -мерным, где K — количество измерений в нашем векторе изображения/признака. Переменная $W_{gradient}$ — это фактический градиент, где у нас есть запись градиента для каждого измерения.

Затем мы применяем градиентный спуск к строке 3. Мы умножаем наш W градиент на альфа (α), что наша скорость обучения. Скорость обучения контролирует размер нашего шага.

На практике выплатите много времени на поиск оптимального значения — это, безусловно, самый важный параметр в вашей модели. Если α слишком велико, вы будете проводить все свое время, прыгая по ландшафту потерь, никогда фактически не «спускаясь» на дно бассейна (если ваши случайные прыжки не приведут вас туда по чистой случайности). И наоборот, если α слишком мало, то потребуется много (возможно, непомерно много) итераций, чтобы достичь дна бассейна. Поиск оптимального значения вовсе у вас много головной боли, и выплатите значительное количество времени, пыгаясь найти оптимальное значение этой переменной для вашей модели и набора данных.

9.1.6 Реализация базового градиентного спуска в Python

Теперь, когда мы знаем основы градиентного спуска, давайте реализуем его в Python и используем для классификации некоторых данных.

Откройте новый файл, назовите его `gradient_descent.py` и вставьте следующий код:

```
1 # импортируем необходимые пакеты
2 из sklearn.model_selection import train_test_split 3 из sklearn.metrics
3 import classification_report 4 из sklearn.datasets import make_blobs 5
4 import matplotlib.pyplot as plt 6 import numpy as np 7 import argparse
5
6
7
8
9 def sigmoid_activation(x): #
10     ВЫСЛИТЬ значение активации сигмоида для данного ввода return
11     1.0 / (1 + np.exp(-x))
```

Строки 2-7 импортируют необходимые пакеты Python. Мы видели все эти импорты раньше, за исключением `make_blobs`, функции, используемой для создания «клякс» нормально распределенных точек данных — это удобная функция при тестировании или реализации наших собственных моделей с нуля.

Затем мы определяем функцию `sigmoid_activation` в строке 9. На графике эта функция будет напоминать S-образную кривую (рис. 9.4). Мы называем эту функцией активации, потому что функция «активируется» и срабатывает «ВКЛ» (весь одно значение > 0.5) или «ВБКЛ» (весь одно значение $<= 0.5$) в зависимости от всех однмых данных x .

Мы можем определить эту связь с помощью метода предсказания ниже:

13 по определению предсказать (X, W):

14 # В озьмем скалярное произведение между нашими функциями и матрицей
15 весов `preds = sigmoid_activation(X.dot(W))`

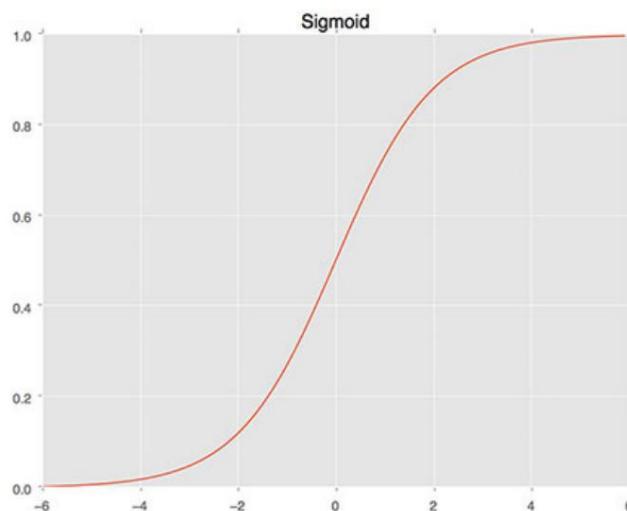


Рисунок 9.4: Сигмовидная функция активации. Эта функция сосредоточена в точках $x = 0,5$, $y = 0,5$. Функция насыщается в x в остах.

```

16
17     # применить ступенчатую функцию для порога двоичных вьюков
18     # ярлыки классов
19     преды[преды<= 0,5] = 0
20     преды[преды> 0] = 1
21
22     # возврашаем предсказания
23     вернуть пред

```

Учитывая набор точек x одних данных X и веса W , мы вызываем функцию `sigmoid_activation` на них, чтобы получить набор прогнозов (строка 15). Затем мы порогуем прогнозы любой прогноз с означением $\leq 0,5$ устанавливается равным 0, а любой прогноз с означением $> 0,5$ устанавливается равным 1 (строки 19 и 20). Затем прогнозы возвращаются в виде функции в строке 23.

X отягчить и другие (лучшие) альтернативы сигмовидной функции активации, отличаясь от правильной точки в нашем обсуждении нейронных сетей, глубокого обучения и градиентного обучения оптимизация. Я буду обсуждать другие функции активации в главе 10 Starter Bundle и Глава 7 комплекта для практикующих, а пока просто имейте в виду, что сигмовидная — это нелинейная функция активации, которую мы можем использовать для порога наших прогнозов.

Далее, давайте проанализируем наши аргументы командной строки:

```

25 # построить аргумент parse и разобрать аргументы
26 ap = argparse.ArgumentParser()
27 ap.add_argument("-e", "--epochs", type=float, default=100,
28                  help="количество эпох")
29 ap.add_argument("-a", "--alpha", type=float, default=0.01,
30                  help="скорость обучения")
31 аргумент = ap.parse_args()

```

Мы можем предоставить нашему сценарию два (необязательных) аргумента командной строки:

- `--epochs`: количество эпох, которые мы будем использовать при обучении нашего классификатора с использованием градиента спуска.

- `--alpha`: Скорость обучения для градиентного спуска. Обычно мы видим 0,1, 0,01 и 0,001 в качестве начальных значений скорости обучения, но опять же, это гиперпараметр, который вам нужно настроить для ваших собственных задач классификации.

Теперь, когда наши аргументы командной строки проанализированы, давайте сгенерируем некоторое данные для классификации:

```

33 # сгенерировать задачу классификации 2-х классов с 1000 точек данных,
34 # где каждая точка данных представляет собой двумерный вектор признаков
35 (X, y) = make_blobs(n_samples=1000, n_features=2, center=2, cluster_std = 1.5,
36                     random_state=1) 37 y = y.reshape((y.shape[0], 1))

38
39 # вставить столбец с единицами в качестве последней записи в матрице
признаков 40 # -- этот маленький трюк позволяет нам обрабатывать систематическую ошибку
41 # как обучаемый параметр в матрице весов 42 X = np.c_[X, np.ones((X.shape[0]))]

43
44 # разбить данные на обучающую и тестовую части, используя 50% от 45 # данные для обучения и
оставшиеся 50% для тестирования 46 (trainX, testX, trainY, testY) = train_test_split(X, y,
47                           test_size=0.5, random_state=42)

```

В строке 35 мы вызываем `make_blobs`, который генерирует 1000 точек данных, разделенных на два класса. Эти точки данных являются двумерными, что означает, что «векторы признаков» имеют длину 2. Метки для каждой из этих точек данных равны 0 или 1. Наша цель — обучить классификатор, который правильно предсказывает метку класса для каждой точки данных.

В строке 42 применяется «трюк смещения» (подробно описаный выше), который позволяет нам пропустить явное отслеживание нашего вектора смещения, вставив совершенно новый столбец единиц в качестве последней записи в нашу матрицу дизайна X. Добавление столбца, содержащего постоянное значение для всех векторов признаков, позволяет нам рассматривать наше смещение как обучаемый параметр в матрице весов W, а не как совершенно отдельную переменную.

После того, как мы вставили столбец единиц, мы разделяем данные на обучение и тестирование. Разбивается на строки 46 и 47, используя 50% данных для обучения и 50% для тестирования.

Наш следующий блок кода обрабатывает случайную инициализацию нашей векторной матрицы с использованием равномерного распределения, чтобы она имела тоже количество измерений, что и наши векторные признаки (включая смещение):

```

49 # инициализируем нашу матрицу весов и список потерь 50 print("[INFO]
training...")

51 В t = np.random.randn(X.shape[1], 1) 52
потери = []

```

Вы также можете увидеть инициализацию как с нулевым, так и с единичным вектором, но, как мы знаем позже в этой книге, хордовая инициализация имеет решающее значение для обучения нейронной сети за разумное время, поэтому случайная инициализация вместе с простой эвристикой побеждают в подавляющем большинстве обстоятельств [84].

Строка 52 инициализирует список для отслеживания наших потерь после каждой эпохи. В конце в скрипта Python мы построим график потерь (которые в идеале должны уменьшаться со временем).

Все наши переменные теперь инициализированы, поэтому мы можем перейти к фактической процедуре обучения и градиентного спуска:

```

54 # цикл по нужному количеству эпох 55 для
эпох и в np.arange(0, args["epochs"]):

```

```

56     # возводим скалярное произведение между нашими функциями 'X' и весом
57     # матрица 'W', затем передаем значение через нашу сигмовидную активацию
58     # функция тем самым давая нам наши прогнозы по набору данных
59     preds = sigmoid_activation(trainX.dot(W))

60
61     # теперь, когда у нас есть наши прогнозы нам нужно определить
62     # 'ошибку', которая является разницей между нашими предсказаниями и
63     # истинные значения
64     ошибка = предс - поезд Y
65     потеря = пр.sum(ошибка ** 2)
66     потеря.добавлять(потери)

```

В строке 55 мы начинаем перебирать заданное количество --epochs. По умолчанию мы разрешаем процедуру обучения чтобы «увидеть» каждую из обучающих точек всего 100 раз (таким образом, 100 эпох).

Строка 59 берет скалярное произведение между всем нашим тренировочным набором trainX и нашей матрицей весов . W. В конце этого скалярного произведения подается через сигмовидную функцию активации, что дает наше предсказания

Учитывая наши прогнозы следующим шагом будет определение «ошибки» прогнозов или более простой разница между нашими прогнозами и истинными значениями (строка 64). Страна 65 вычисляет ошибку наименьших квадратов по нашим прогнозам, приставив потерю обычной используемой для машинной классификации проблемы. Цель этой процедуры обучения — минимизировать нашу ошибку метода наименьших квадратов. Мы добавляем этот убыток в наш список убытков в строке 66, чтобы позже мы могли отобразить убыток с течением времени.

Теперь, когда у нас есть ошибка, мы можем вычислить градиент , а затем использовать его для обновления нашего вектора матрицы W:

```

68     # обновление градиентного спуска является скалярным произведением между нашим
69     # особенностями и ошибкой прогнозов
70     градиент = trainX.T.dot(ошибка)

71
72     # на этапе обновления важно, что нам нужно сделать, это "подтолкнуть" в вес
73     # матрица в отрицательном направлении градиента (отсюда
74     # термин "градиентный спуск", сделав небольшой шаг к набору
75     # "более оптимальных" параметров
76     W += -args["альфа"] * градиент

77
78     # проверяем, должны ли обновляться обновление
79     если epoch % 5 == 0:
80         print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
81             потеря))

```

Строка 70 обрабатывает градиент, который является точечным произведением между нашими точками данных X и ошибкой.

Строка 76 — самый важный шаг в нашем алгоритме, именно здесь происходит фактический градиентный спуск. Здесь мы обновляем нашу весовую матрицу W , делая шаг в отрицательном направлении градиента, тем самым позволяя нам двигаться к нижней части бассейна ландшафта потерь (отсюда термин градиентный спуск). После обновления нашей матрицы весов мы проверяем, следует ли обновлять или обновляется на нашем терминале (строки 79-81) , а затем продолжает цикл до желаемого количества эпох было выполнено - градиентный спуск, таким образом, является итеративным алгоритмом.

Теперь наш классификатор обучен. Следующий шаг — оценка:

83 # оценить нашу модель

84 print("[INFO] оценка...")

```
85 preds = прогноз (testX, W) 86
print (classification_report (testY, preds))
```

Чтобы на самом деле делать прогнозы и используя нашу матрицу весов W , мы ввиваем метод прогнозирования для $testX$ и W в строке 85. Учитывая прогнозы мы отображаем с помощью отформатированного отчета о классификации на нашем терминале в строке 86.

Наш последний блок кода обрабатывает график (1) данных тестирования чтобы мы могли визуализировать набор данных, который мы пыгались классифицировать и (2) наши потери с течением времени:

```
88 # построить (тестирование) классификационные
данные 89 plt.style.use("ggplot") 90 plt.figure() 91
plt.title("Data") 92 plt.scatter(testX[:, 0], testX[:, 1],
маркер="o", c=testY, s=30)
```

93

```
94 # построить график, отображающий потери с течением
времени _ plt.title("Потери при обучении") 99 plt.xlabel("Эпох а
№") 100 plt.ylabel("Потери") 101 plt.show()
```



9.1.7 Результаты простого градиентного спуска

Чтобы

выполнить наш скрипт, просто введите следующую команду:

```
$ python gradient_descent.py
[INFO] обучение...
[INFO] Эпох а=1, потерь=486,5895513
[INFO] Эпох а=5, потерь=11,1087812
[INFO] Эпох а=10, потерь=9,1312984
[INFO] Эпох а=15, потерь=7,0049498
[INFO] Эпох а=20, потерь=6,9914949
[INFO] Эпох а=25, потерь=6,9382765
[INFO] Эпох а=30, потерь=5,8285461
[INFO] Эпох а=35, потерь=4,1750536
[INFO] Эпох а=40, потерь=2,7319634
[INFO] Эпох а=45, потерь=1,3891531
[INFO] Эпох а=50, потерь=1,0787792
[INFO] Эпох а=55, потерь=0,8927193
[INFO] Эпох а=60, потерь=0,6001450
[INFO] Эпох а=65, потерь=0,3200953
[INFO] Эпох а=70, потерь=0,1651333
[INFO] Эпох а=75, потерь=0,0941329
[INFO] Эпох а=80, потерь=0,0602669
[INFO] Эпох а=85, потерь=0,0424516
[INFO] Эпох а=90, потерь=0,0321485
[INFO] Эпох а=95, потерь=0,0256970
[INFO] Эпох а = 100, потерь= 0,0213877
```

Как видно из рис. 9.5 (слева), наш набор данных явно линейно разделим (т. е. мы можем провести линию, разделяющую два класса данных). Наши потери также резко падают, начиная с очень высоких

а затем быстро падает (справа). Мы можем увидеть, насколько быстрые сокращаются потери, исследуя вьевод терминала выше. Обратите внимание, что потери изначально > 400 , но падают до 1,0 к эпохе 50. Время, когда обучение заканчивается эпохе 100, наши потери упали на порядок до 0,02.

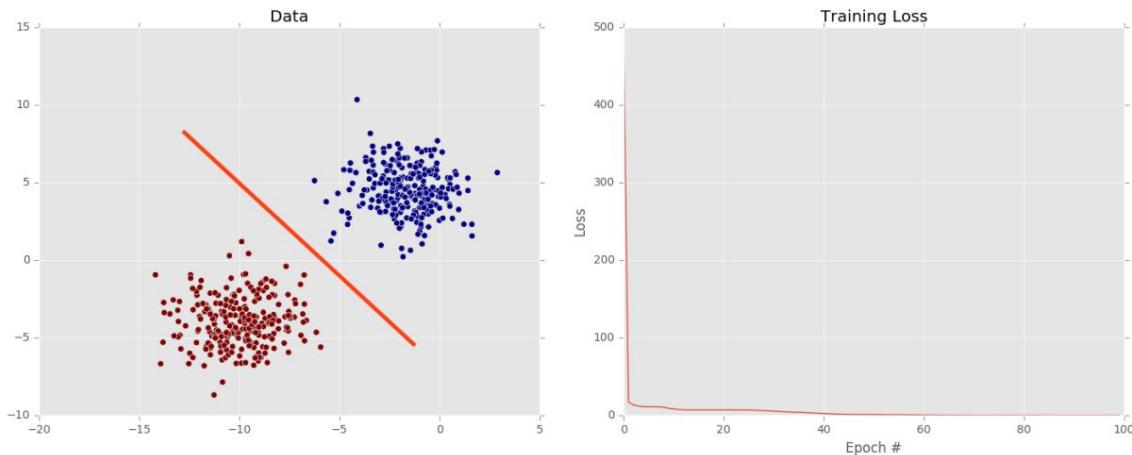


Рисунок 9.5: Слева: вх однай набор данных, который мы пыгаемся разделить на два набора: красный и синий. Этот набор данных явно линейно разделим, так как мы можем провести одну линию, которая аккуратно делит набор данных на два класса. Справа: изучение набора параметров для классификации нашего набора данных с помощью градиентного спуска. Потеря начинается очень высоко, но быстро падает почти до нуля.

Этот график подтверждает, что наш весовая матрица обновляется таким образом, что позволяет классификатору учиться на обучающих данных. Однако, основываясь на оставшейся части вьюода нашего терминала, кажется что наш классификатор неправильно классифицировал несколько точек данных (< 5 из них):

[INFO] оценка...				
	точность	вспомнить поддержку f1-score		
0	1,00	0,99	1,00	250
1	0,99	1,00	1,00	250
среднее / общее	1,00	1,00	1,00	500

Обратите внимание, как нулевой класс классифицируется правильно в 100% случаев, но один класс классифицируется правильно только в 99% случаев. Причина этого не соответствия в том, что ванильный градиентный спуск выполняет обновление веса только один раз для каждой эпохи — в этом примере мы обучили нашу модель для 100 эпох, поэтому произошло только 100 обновлений. В зависимости от инициализации весовой матрицы и размера скорости обучения, возможно, мы не сможем изучить модель, которая может разделить точки (даже если они линейно отдельны).

Фактически, последующие запуски этого скрипта могут показать, что оба класса могут быть правильно классифицированы 100% времени — результат зависит от начальных значений, которые принимает W . Чтобы проверить этот результат самостоятельно, запустите скрипт `gradient_descent.py` несколько раз.

Для простого градиентного спуска вам лучше тренироваться для большего количества эпох с меньшим обучением. скорость, чтобы помочь решить эту проблему. Однако, как мы видим в следующем разделе, вариант градиента спуск под названием Stochastic Gradient Descent выполняет обновление веса для каждой партии обучения данных, подразумевая что за эпоху происходит несколько обновлений веса. Такой подход приводит к более быстрому и устойчивым результатам.

9.2 Стохастический градиентный спуск (SGD)

В предыдущем разделе мы обсудили градиентный спуск, алгоритм оптимизации первого порядка, который можно использовать для изучения набора весов классификатора для параметризованного обучения. Однако эта «ванильная» реализация градиентного спуска может быть чрезмерно медленной для работы с большими наборами данных — на самом деле, ее даже можно считать вычислительной расточительностью.

Вместо этого мы должны применить стохастический градиентный спуск (SGD), простую модификацию стандартного алгоритма градиентного спуска, который вычисляет градиент и обновляет весовую матрицу W для небольших пакетов обучающих данных, а не для всего обучающего набора. Хотя эта модификация приводит к «более шумным» обновлениям, она также позволяет нам делать больше шагов по градиенту (один шаг на каждую партию по сравнению с одним шагом на эпоху), что в конечном итоге приводит к более быстрой сходимости и не оказывает негативного влияния на потерю и точность классификации.

SGD, пожалуй, самый важный алгоритм, когда речь идет об обучении глубоких нейронных сетей. Несмотря на то, что первоначальное описание SGD было представлено более 57 лет назад [85], он по-прежнему является движителем, который позволяет нам обучать большие сети для изучения закономерностей из точек данных. Помимо всех других алгоритмов, описанных в этой книге, уделите время изучению SGD.

9.2.1 Мини-пакет SGD

Рассматривая алгоритм ванильного градиентного спуска, должно быть (в некоторой степени) очевидно, что этот метод будет работать очень медленно на больших наборах данных. Причина этой медленности заключается в том, что каждая итерация градиентного спуска требует от нас вычисления прогноза для каждой точки обучения из наших обучающих данных, прежде чем нам будет разрешено обновить нашу матрицу весов. Для наборов данных изображений, таких как ImageNet, где у нас есть более 1,2 миллиона обучающих изображений, это вычисление может занять много времени.

Также оказывается, что вычисление прогнозов для каждой точки обучения перед тем, как сделать шаг вперед, наша весовая матрица расточительна в вычислительном отношении и мало что делает для покрытия нашей модели.

Вместо этого мы должны пакетировать наши обновления. Мы можем обновить псевдокод, чтобы преобразовать ванильный градиентный спуск в SGD, добавив дополнительный вектор функции:

```
1 , x отяверю:
2     пакет = next_training_batch (данье, 256)
3     Wградиент = оценка_градиента (потеря, партия Вт)
4     W += -альфа * Wградиент
```

Единственная разница между ванильным градиентным спуском и SGD заключается в добавлении `next_training_batch`. Функция в местного, чтобы вычислить наш градиент по всему набору данных, мы вместо этого выбираем наши данные, получая пакет. Мы оцениваем градиент для партии и обновляем нашу матрицу весов W . Сточки зрения реализации мы также пыгаемся рандомизировать наши обучающие выборки перед применением SGD, поскольку алгоритм чувствителен к партиям.

Взглянув на псевдокод для SGD, вы сразу заметите введение нового параметра: размер пакета. В «чистой» реализации SGD размер вашего мини-пакета будет равен 1, что означает, что мы будем случайным образом выбирать одну точку данных из обучающего набора, вычислять градиент и обновлять наши параметры. Однако мы часто используем мини-пакеты, которые > 1 . Типичные размеры пакетов включают 32, 64, 128 и 256.

Итак, зачем использовать размер партии > 1 ? В первых, размеры пакетов > 1 помогают уменьшить дисперсию при обновлении параметров (<http://pyimg.co/pd5w0>), что приводит к более стабильной сходимости. Во вторых, степень двойки часто желательна для размеров пакетов, поскольку они позволяют встроенным библиотекам оптимизации линейной алгебры быть более эффективными.

В общем, размер мини-пакета — это не тот гиперпараметр, о котором следует беспокоиться [57]. Если вы используете графический процессор для обучения нейронной сети, вы определяете, сколько обучающих примеров поместится в ваш графический процессор, а затем используете ближайшую степень двойки в качестве размера пакета, чтобы пакет

поместится на GPU. Для обучения ЦПобынно используется один из перечисленных выше размеров пакетов, чтобы гарантировать выполнение задачи оптимизации линейной алгебры.

9.2.2 Реализация мини-пакета SGD

Давайте продолжим реализацию SGD и посмотрим, чем он отличается от стандартного ванильного градиентного спуска. Откройте новый файл, назовите его sgd.py и вставьте следующий код:

```

1 # импортируем необходимые пакеты
2 из sklearn.model_selection импорт train_test_split
3 из sklearn.metrics импорт classification_report_
4 из sklearn.datasets импортировать make_blobs
5 импортировать matplotlib.pyplot как plt
6 импортировать numpy как np
7 импортировать сингаксический анализ
8
9 поопределению sigmoid_activation(x):
10     # вычислить значение активации сигмоиды для заданного ввода
11     вернуть 1,0 / (1 + np.exp(-x))

```

Строки 2-7* импортируют наши необходимые пакеты Python, точно такие же, как и в файле gradient_descent.py. Пример ранее в этой главе. Строки 9-11 определяют функцию sigmoid_activation, которая также идентична предыдущей версии градиентного спуска.

На самом деле, метод предсказания тоже не меняется

```

13 поопределению предсказать(X, W):
14     # возвратим скалярное произведение между нашими функциями и матрицей весов
15     preds = sigmoid_activation(X.dot(W))

16
17     # применить ступенчатую функцию для порога двоичных выходов
18     # ярлыки классов
19     преды[преды <= 0,5] = 0
20     преды[преды > 0] = 1
21
22     # возвратим предсказания
23     вернуть пред

```

Однако что изменилось, так это добавление функции next_batch:

```

25 поопределению next_batch(X, у, размер_пакета):
26     # перебираем наш набор данных 'X' в мини-пакетах, получая кортеж
27     # текущие пакетные данные и метки
28     для i в пр.arange(0, X.shape[0], batchSize):
29         въект(X[i:i + размер партии], у[i:i + размер партии])

```

Метод next_batch требует трех параметров:

1. X: наш набор данных векторов признаков/инженерных свойств небообработанных пикселей изображения
2. у: метки классов, связанные с каждой из точек обучения соответствующих данных.
3. batchSize: Размер каждого мини-пакета, который будет возвращен

Строки 28 и 29 затем перебирают обучающие примеры получая подмножество X и у как мини-пакеты.

Далее мы можем проанализировать наши аргументы командной строки:

```

31 # построить аргумент parse и разобрать аргументы
32 ap = argparse.ArgumentParser()
33 ap.add_argument("-e", "--epochs", type=float, default=100,
34                 help="количество эпох")
35 ap.add_argument("-a", "--alpha", type=float, default=0.01,
36                 help="скорость обучения")
37 ap.add_argument("-b", "--batch-size", type=int, по умолчанию=32,
38                 help="размер мини-пакетов SGD")
39 аргументов = вары(ap.parse_args())

```

Мы уже рассмотрели параметры --epochs (количество эпох) и --alpha (скорость обучения).
переключитесь с примера с ванильным градиентным спуском, но также обратите внимание, что мы представляем третий переключатель: --batch-size, что, как следует из названия, является размером каждого из наших мини-пакетов. Мы по умолчанию это значение должно составлять 32 точки данных на мини- партию.

Наш следующий блок кода обрабатывает задачу классификации 2 классов с 1000 данными.
баллы добавив столбец смещения а затем выполнив разделение обучения и тестирования

```

41     # создать задачу классификации 2 классов с 1000 точек данных,
42 # где каждая точка данных представляет собой двумерный вектор признаков
43 (X, y) = make_blobs(n_samples=1000, n_features=2, center= 2,
44                      cluster_std=1.5, random_state=1)
45 y = y.reshape(y.shape[0], 1)
46
47 # вставить столбец из 1 в качестве последней записи в функции
48 # матрица -- эта маленькая хитрость позволяет нам лечить смещение
49 # как обучаемый параметр в весовой матрице
50 X = np.c_[X, np.ones((X.shape[0]))]
51
52 # разбить данные на тренировочную и тестовую части, используя 50%
53 # данные для обучения и остальные 50% для тестирования
54 (trainX, testX, trainY, testY) = train_test_split(X, y,
55                                                 test_size=0.5, random_state=42)

```

Затем мы инициализируем нашу матрицу весов и потерь, как в предыдущем примере:

```

57 # инициализируем нашу матрицу весов и список потерь
58 print("[ИФ О обучение...]")
59 В t = np.random.randn(X.shape[1], 1)
60 потерь = []

```

Настоящее изменение происходит затем, когда мы перебираем нужное количество эпох, сэмплируя мини- партии по пути:

```

62 # перебрать желаемое количество эпох
63 для эпох в np.arange(0, args["эпох и"]):
64     # инициализируем общие потери за эпоху
65     эпох аПотерь = []
66
67     # перебираем наши данные партиями
68     для(batchX, batchY) в next_batch(X, y, args["batch_size"]):
69         # возвращаем скалярное произведение между нашим текущим набором функций

```

```

70         # и матрицу весов, затем передаем это значение через наш
71         # функция активации
72         preds = sigmoid_activation(batchX.dot(W))
73
74         # теперь, когда у нас есть наши прогнозы нам нужно определить
75         # 'ошибку', которая представляет собой разницу между нашими предсказаниями
76         # и истинные значения
77         ошибка = Предыдущие - партия/
78         epochLoss.append(np.sum(ошибка ** 2))

```

В строке 63 мы начинаем перебирать заданное количество epochs. Затем мы перебираем наш обучающие данные партиями в строке 68. Для каждой партии мы вычисляем скалярное произведение между партиями и W, затем передайте результат через сигмовидную функцию активации, чтобы получить наши прогнозы. Мы вычислить ошибку наименьших квадратов для пакета в строке 77 и использовать это значение для обновления нашего epochLoss на линии 78.

Теперь, когда у нас есть ошибка, мы можем выполнить обновление градиентного спуска, которое является точкой продукт между текущими точками данных партии и ошибкой в партии:

```

80         # обновление градиентного спуска является скалярным произведением между нашим
81         # текущий пакет и ошибка в пакете
82         градиент = пакет.X.T.dot(ошибка)
83
84         # на этапе обновления все, что нам нужно сделать, это "подтолкнуть"
85         # в весовую матрица в отрицательном направлении градиента
86         # (отсюда и термин "градиентный спуск"), сделав небольшой шаг
87         # в сторону набора "более оптимальных" параметров
88         W += -args["альфа"] * градиент

```

Строка 88 обрабатывает обновление нашей матрицы весов на основе градиента, масштабированного по нашей скорости обучения -альфа. Обратите внимание, как этап обновления веса происходит в пакетном цикле — это означает, что несколько обновлений веса за эпоху.

Затем мы можем обновить нашу историю потерь, взяв среднее значение по всем партиям в эпоху и затем отображение обновления на нашем терминале, если это необходимо:

```

90         # обновить нашу историю убытков, взяв средний убыток по всем
91         # партии
92         потеря= np.average(epochLoss)
93         потеря.добавлять(потери)
94
95         # проверяем, должны ли отображаться обновление
96         если эпох a == 0 или (эпох a + 1) % 5 == 0:
97             print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
98                                         потеря))

```

Оценка нашего классификатора выполняется так же, как и в ванильном градиентном спуске — просто въявите и прогнозировать данные testX, используя нашу изученную матрицу весов W:

```

100 # оценить нашу модель
101 print("[INFO] оценка...")
102 предс = предсказать(testX, W)
103 print(classification_report(testY, предс))

```

Мы закажем наш сценарий, нарисуем график **данные** классификации тестирования а также потери за эпохи:

```

105 # построить данные классификации (тестирования)
106 plt.style.use("ggplot")
107 plt.figure()
108 plt.title("Данные")
109 plt.scatter(testX[:, 0], testX[:, 1], marker="o", c=testY, s=30)
110
111 # построить график, отображающий потери в течение времени
112 plt.style.use("ggplot")
113 plt.figure()
114 plt.plot(np.arange(0, args["эпох и"]), потери)
115 plt.title("Потери на тренировке")
116 plt.xlabel("Эпох и #")
117 plt.ylabel("Убыток")
118 plt.show()

```

9.2.3 Результаты SGD

Чтобы изучить результаты нашей реализации, просто выполните следующую команду:

```

$ питонsgd.py
[INFO] обучение...
[INFO] эпох а = 1, потеря = 0,3701232
[INFO] эпох а = 5, потеря = 0,0195247
[INFO] эпох а = 10, потеря = 0,0142936
[INFO] эпох а = 15, потеря = 0,0118625
[INFO] эпох а = 20, потеря = 0,0103219
[INFO] эпох а = 25, потеря = 0,0092114
[INFO] эпох а = 30, потеря = 0,0083527
[INFO] эпох а = 35, потеря = 0,0076589
[INFO] эпох а = 40, потеря = 0,0070813
[INFO] эпох а = 45, потеря = 0,0065899
[INFO] эпох а = 50, потеря = 0,0061647
[INFO] эпох а = 55, потеря = 0,0057920
[INFO] эпох а = 60, потеря = 0,0054620
[INFO] эпох а = 65, потеря = 0,0051670
[INFO] эпох а = 70, потеря = 0,0049015
[INFO] эпох а = 75, потеря = 0,0046611
[INFO] эпох а = 80, потеря = 0,0044421
[INFO] эпох а = 85, потеря = 0,0042416
[INFO] эпох а = 90, потеря = 0,0040575
[INFO] эпох а = 95, потеря = 0,0038875
[INFO] эпох а = 100, потеря = 0,0037303
[INFO] оценка...

      Точность      Вспоминать поддержку f1-score

        0           1,00           1,00           1,00          250
        1           1,00           1,00           1,00          250

    среднее / общее       1,00           1,00           1,00          50

```

Мы будем использовать для классификации тот же набор данных «блоб», что и на рис. 9.5 (слева) выше, чтобы мы могли сравнить наши результаты SGD с ванильным градиентным спуском. Кроме того, пример SGD использует тот же

скорость обучения(0,1) и такое же количество эпох (100), как у ванильного градиентного спуска. Однако обратите внимание, насколько сглажена наша кривая потерь на рис. 9.6.

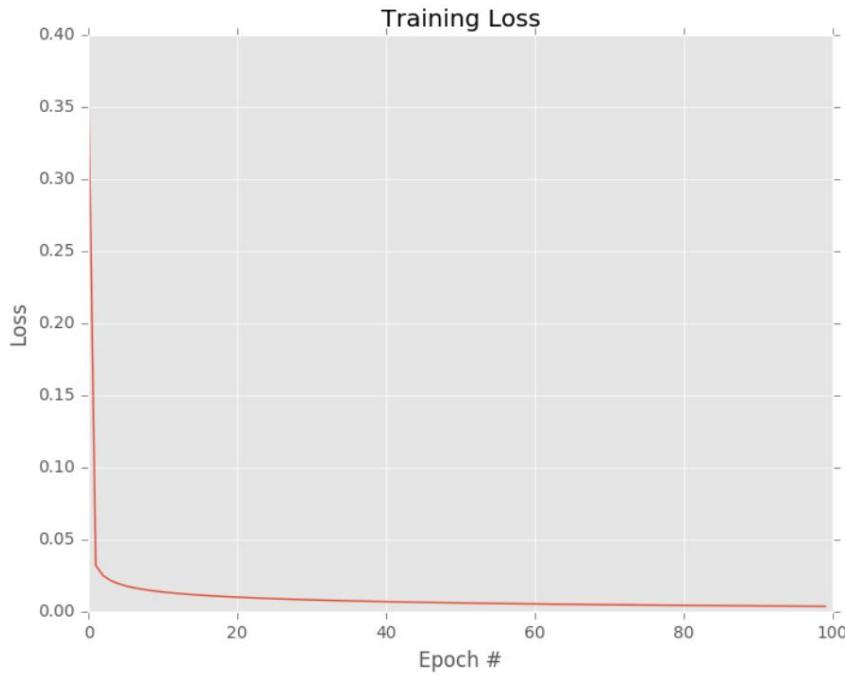


Рисунок 9.6: Применение стохастического градиентного спуска к нашему набору данных красных и синих точек данных. Используя SGD, наша кривая обучения становится намного более плавной. Кроме того, мы можем получить на порядок меньшие потери к концу 100-й эпохи (по сравнению со стандартным ванильным градиентным спуском).

Исследуя фактические значения потерь в конце 100-й эпохи и, в частности, что потери, полученные с помощью SGD, на порядок ниже, чем при ванильном градиентном спуске (0,003 против 0,021 соответственно). Эта разница связана с несколькими обновлениями весов за эпоху, что дает нашей модели больше шансов учиться на обновлениях, внесенных в матрицу весов. Этот эффект еще более заметен на больших наборах данных, таких как ImageNet, где у нас есть миллионы обучающих примеров, а небольшие добавочные обновления наших параметров могут привести к решению с низкими потерями (но не обязательно оптимальному).

9.3 Расширения для SGD

На практике вы столкнетесь с двумя основными расширениями SGD. Первый — это импульс [86], метод, используемый для ускорения SGD, что позволяет ему быстрее обучаться, фокусируясь на измерениях, градиент которых направлен в одном направлении. Второй метод — ускорение Нестерова [87], расширение стандартного импульса.

9.3.1 Импульс

Вспомните свою любимую детскую игровую площадку, где вы проводили дни, катаясь с холма, покрывая себя травой и грязью (к большому горючению вашей матери). Помимо того, как вы спускаетесь с холма, вы создаете все больший и больший импульс, который, в свою очередь, несет вас быстрее вниз по склону.

Импульс, примененный к SGD, имеет тот же эффект — наша цель состоит в том, чтобы опираться на стандартное обновление веса, чтобы включить термин импульса, что позволит нашей модели получить меньшие потери (и более высокие значения).

точность) за меньшее количество эпох. Следовательно, импульс должен увеличивать силу обновлений для измерений, градиенты которых указывают в одном и том же направлении, а затем уменьшать силу обновлений для измерений, градиенты которых меняют направление [86, 88].

Наше предыдущее правило обновления веса простое: ключом масштабирования градиента по нашей скорости обучения

$$W = W - \alpha \cdot W f(W) \quad (9.4)$$

Теперь введем импульсный член V , масштабированный по γ :

$$V = \gamma V - \alpha \cdot W f(W) \quad B_t = B_t + V \quad (9.5)$$

Член импульса обычно устанавливается равным 0,9; хотя другой распространенной практикой является установка на 0,5 до тех пор, пока обучение не стабилизируется, а затем увеличение его до 0,9 — крайне редко можно увидеть импульс <0,5. За более подробным обзором импульса обратитесь к Sutton [89] и Qian [86].

9.3.2 Ускорение Нестерова

Предположим, вы вернулись на свою детскую игровую площадку, катаясь с горки. Вы создали импульс и двигаетесь довольно быстро, но есть проблема. У подножия олма находитесь кирпичная стена вашей школы в которую выбыли не врезаться на полной скорости.

Та же мысль может быть применена к SGD. Если мы создадим слишком большой импульс, мы можем выйти за пределы локального минимума и продолжить движение. Поэтому было бы выгодно иметь более умный крен, который знает, когда нужно замедлиться и здесь вступает ускоренный градиент Нестерова [87].

Ускорение Нестерова можно представить как корректирующее обновление импульса, которое позволяет нам получить приблизительное представление от того, где будут наши параметры после обновления. Глядя на «Обзор слайдов мини-пакетного градиентного спуска» Хингана [90], мы можем увидеть хорошую визуализацию ускорения Нестерова (рис. 9.7).

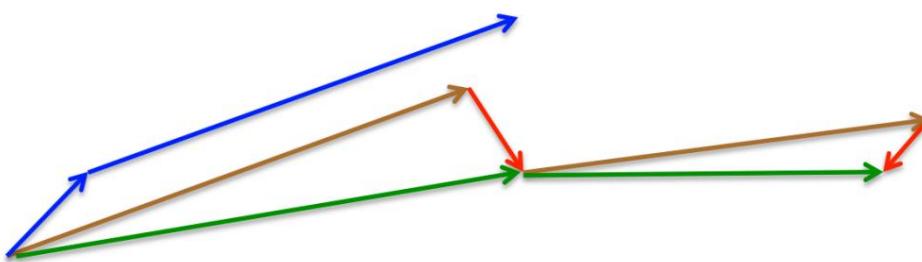


Рисунок 9.7: Графическое изображение ускорения Нестерова. Сначала мы делаем большой скачок в направлении предыдущего градиента, затем измеряем градиент там, где мы оказались, и вносим поправку.

Используя стандартный импульс, мы вычисляем градиент (маленький синий вектор), а затем делаем большой скачок в направлении градиента (большой синий вектор). При нестеровском ускорении мы сначала делаем большой скачок в направлении нашего предыдущего градиента (коричневый вектор), измеряем градиент, а затем делаем коррекцию (красный вектор) — зеленый вектор является окончательно скорректированным обновлением по нестеровскому ускорению (переформулировано Рудером [88]).

Тщательное теоретическое и математическое рассмотрение нестеровского ускорения выходит за рамки этой книги. Для тех, кто заинтересован в более подробном изучении нестеровского ускорения, пожалуйста, обратитесь к Рудеру [88], Бениоу [91] и Суцкеверу [92].

9.3.3 Отдельные рекомендации

Импульс — важный термин, который может увеличить сходимость нашей модели; мы как правило, не беспокоимся об этом гиперпараметре так сильно, как нашей скорости обучения или штрафе за регуляризацию (обсуждаемом в следующем разделе), которые, безусловно, являются наиболее важными регуляторами для настройки.

Мое личное эмпирическое правило заключается в том, что всякий раз, когда вы используете SGD, также применяйте импульс. В большинстве случаев вы можете установить (и оставить) 0,9, хотя Karpathy [93] предлагает начинать с 0,5 и увеличивать его для больших значений по мере увеличения эпох.

Что касается ускорения Нестерова, я бы не использовал его для небольших наборов данных, но для больших наборов данных (таких как ImageNet) я его почти всегда избегаю. В то время как ускорение Нестерова имеет надежные теоретические гарантии, все основные публикации, обучение на ImageNet (например, AlexNet [94], VGGNet [95], ResNet [96], Inception [97] и т. д.), используют SGD с импульсом — ни одна статья из этих основ опирается на группу, использующую ускорение Нестерова.

Мой личный опыт привел меня в выводу, что при обучении глубоких сетей на больших наборах данных с SGD легче работать при использовании импульса и исключении ускорения Нестерова. С другой стороны, меньшие наборы данных, как правило, пользуются преимуществами ускорения Нестерова.

Однако имейте в виду, что это мое анекдотическое мнение и что ваша пробег может отличаться.

9.4 Регуляризация

«Многие стратегии, используемые в машинном обучении, специально разработаны для уменьшения ошибки теста, возможно, за счет увеличения ошибки обучения. Эти стратегии в совокупности известны как регуляризация». — Гудфеллоу и др. [10]

В предыдущих разделах этой главы мы обсуждали две важные функции потерь: потери мультиклассового SVM и кросс-энтропийные потери. Затем мы обсудили градиентный спуск и то, как сеть может учиться обновлять весовье параметры модели. Хотя наша функция потерь позволяет нам определить, насколько хорошо (или плохо) наш набор параметров выполняет данную задачу классификации, сама функция потерь не учитывает, как «выглядит» весовая матрица.

Что я подразумеваю под «внешностью»? Что ж, имейте в виду, что мы работаем в вещественном пространстве, поэтому существует бесконечный набор параметров, который обеспечивает разумную точность классификации в нашем наборе данных (для каждого определения «разумного»).

Как нам выбрать набор параметров, которые помогут обеспечить хорошее обобщение нашей модели? Или, по крайней мере, уменьшить последствия переобучения. Ответ — регуляризация. Уступая только скорости обучения, регуляризация является наиболее важным параметром в нашей модели, который вы можете настроить.

Существуют различные типы методов регуляризации, такие как регуляризация L1, регуляризация L2 (обычно называемая «уменьшением веса») и Elastic Net [98], которые используются путем обновления самой функции потерь, добавления дополнительного параметра для ограничения пропускной способности модели.

У нас также есть типы регуляризации, которые можно добавить в сетевую архитектуру — отсев является типичным примером такой регуляризации. Затем у нас есть неявные формы регуляризации, которые применяются в процессе обучения. Примеры неявной регуляризации включают увеличение данных и раннюю остановку. В этом разделе мы рассмотрим различные параметризованные регуляризации, полученные путем изменения наших функций потерь и обновления.

В главе 11 Starter Bundle мы рассмотрим отсев, а затем в главе 17 мы более подробно обсудим переобучение, а также то, как мы можем использовать раннюю остановку в качестве регуляризатора. В пакете Practitioner Bundle вы найдете примеры увеличения данных, используемые в качестве регуляризации.

9.4.1 Чем такое регуляризация и зачем она нам нужна?

Регуляризация помогает нам контролировать мощность нашей модели, гарантируя что наши модели лучше выполняют (правильную) классификацию точек данных, на которых они не обучались, что мы называем

умение обобщать. Если мы не применим регуляризацию, наши классификаторы могут легкостать слишком сложными и неподходящими для наших обучающих данных, и в этом случае мы потеряем способность обобщать наши тестовые данные (а также точки данных в тестовом наборе, такие как новые изображения). в дикой природе).

Тем не менее, слишком много регуляризации может быть плохой в вещах. Мы можем столкнуться с риском недообучения и в этом случае наша модель плохо оработает с обучающими данными и не может смоделировать взаимосвязь между вх одными данными и метками вх однокласса (потому что мы слишком сильно ограничили возможности модели). Например, рассмотрим следующий график точек, а также различные функции, соответствующие этим точкам (рис. 9.8).

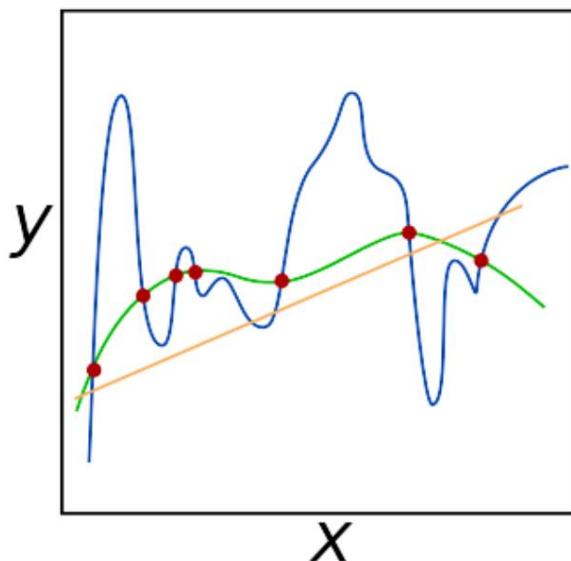


Рисунок 9.8: Пример недообучения (оранжевая линия), переобучения (синяя линия) и обобщения (зеленая линия). Наша цель при создании классификаторов глубокого обучения состоит в том, чтобы получить эти типы «зеленых функций», которые хорошо соответствуют нашим обучающим данным, но избегают переобучения. Регуляризация может помочь нам получить этот тип желаемого соответствия.

Оранжевая линия является примером недообучения — мы не фиксируем взаимосвязь между точками. С другой стороны синяя линия является примером переобучения — у нас слишком много параметров в нашей модели, и хотя она охватывает все точки в наборе данных, она также сильно различается между точками. Это не гладкая простая посадка, которую мы выбрали. Затем у нас есть зеленая функция, которая также затрагивает все точки в нашем наборе данных, но делает это гораздо более предсказуемым и простым способом.

Цель регуляризации состоит в том, чтобы получить эти типы «зеленых функций», которые хорошо соответствуют нашим обучающим данным, но избегают переобучения нашим обучающим данным (синий) или неспособности моделировать базовые отношения (желтый). В главе 17 мы будем видеть, как отслеживать обучение и выявлять недообучение и переобучение; однако на данный момент просто поймите, что регуляризация является критически важным аспектом машинного обучения и мы используем регуляризацию для управления обобщением модели. Чтобы понять регуляризацию и ее влияние на нашу функцию потерь и правило обновления веса, давайте перейдем к следующему разделу.

9.4.2 Обновление нашего обновления по потерям весу для включения регуляризации

Начнем с нашей функции кросс-энтропийных потерь (раздел 8.2.3):

$$L_i = -\frac{1}{N} \sum_{j=1}^H \log(e^{y_j} / e_{\text{сж}}) \quad (9.6)$$

Потери на всем тренировочном наборе можно записать в виде:

$$L = -\frac{1}{N} \sum_{i=1}^H L_i \quad (9.7)$$

Теперь предположим, что мы получили матрицу весов W такую, что каждая точка данных в нашем обучении попала в правильную классификацию, а значит, наша потеря $L = 0$ для всех L_i .

Удивительно, мы получаем 100% точность — но позвольте мне задать вам вопрос об этой векторной матрице — она уникальна? Или, другими словами, есть ли лучший выбор W , который улучшит способность нашей модели обобщать и уменьшить переоснащение?

Если есть такой W , как мы знаем? И как мы можем включить этот тип штрафа в нашу функцию потерь? Ответ заключается в определении штрафа за регуляризацию, функции, которая работает с нашей матрицей весов. Штраф за регуляризацию обычно записывается как функция $R(W)$. Уравнение 9.8 ниже показывает наиболее распространенный штраф за регуляризацию, регуляризацию L2 (также называемую уменьшением веса):

$$R(W) = \frac{1}{2} \sum_{j=1}^H \|w_j\|^2 \quad (9.8)$$

Что именно делает функция? С точки зрения языка Python, он просто берет сумму квадратов каждого элемента массива:

```

1 штраф = 0
2
3 для i в np.arange(0, W.shape[0]): для j в
4     np.arange(0, W.shape[1]): штраф += (W[i][j]
5             ** 2)

```

Здесь мы перебираем все элементы матрицы в цикле суммы квадратов. Сумма квадратов в штрафе за регуляризацию L2 препятствует использованию больших весов в нашей матрице W , предпочитая меньшие. Почему мы можем отдать предпочтение большими значениям веса? Короче говоря, штрафует большие веса, мы можем улучшить способность к обобщению и тем самым уменьшить переоснащение.

Подумайте об этом так: чем большее значение веса, тем большее влияние оно оказывает на весь один прогноз. Измерениями большими значениями веса могут почти единолично контролировать весь один прогноз классификатора (конечно, при условии, что значение веса достаточно велико), что почти наверняка приведет к переобучению.

Чтобы смягчить влияние различных измерений на нашу одинаковую классификацию, мы применяем регуляризацию, тем самым ищем значения W , которые учитывают все измерения и не несколько с большими значениями. На практике вы можете обнаружить, что регуляризация немногого снижает точность обучения, но на самом деле повышает точность тестирования.

Опять же, наша функция потерь имеет ту же базовую форму, только теперь мы добавляем регуляризацию:

$$L = -\frac{1}{N} \sum_{i=1}^H L_i + \lambda R(W) \quad (9.9)$$

Первое слагаемое, которое мы видели раньше, — это средняя потеря в всем образцам в нашей обучющей выборке.

Второй член новый — это наш штраф за регуляризацию. Переменная λ — это гиперпараметр, который управляет количеством или силой применяемой нами регуляризации. На практике и скорость обучения, и член регуляризации λ — это гиперпараметры настройки которых вы можете отрегулировать больше в сегмент времени.

Расширение кросс-энтропийных потерь для классификации регуляризации L2 дает следующее уравнение:

$$L = \frac{1}{N} \sum_{i=1}^N \left[-\log(e^{s_i / e_{s_j}}) \right] + \lambda \sum_{j=1}^J \|w_j\|^2 \quad (9.10)$$

Мы также можем расширить потери Multi-class SVM:

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq s_i} [\max(0, s_j - s_i + 1)] + \lambda \sum_{j=1}^J \|w_j\|^2 \quad (9.11)$$

Теперь давайте взглянем на наше стандартное правило обновления веса:

$$W = W - \alpha \nabla f(W) \quad (9.12)$$

Этот метод обновляет наши веса на основе градиента, умноженного на скорость обучения. Принимая во внимание регуляризации правило обновления веса принимает вид:

$$W = W - \alpha \nabla f(W) + \lambda R(W) \quad (9.13)$$

Здесь мы добавляем отрицательный линейный член к нашим градиентам (т. е. градиентный спуск), штрафуя большие веса, с конечной целью облегчить обобщение нашей модели.

9.4.3 Типы методов регуляризации

В общем, вы видите три распространенных типа регуляризации, которые применяются непосредственно к функции потерь. Первый, который мы рассмотрели ранее, регуляризация L2 (также известная как «распад веса»):

$$R(W) = \sum_{i,j} \|w_{ij}\|^2 \quad (9.14)$$

У нас также есть регуляризация L1, которая принимает абсолютное значение, а не квадрат:

$$R(W) = \sum_{i,j} |w_{ij}| \quad (9.15)$$

Регуляризация Elastic Net [98] стремится объединить регуляризацию как L1, так и L2:

$$R(W) = \sum_{i,j} \beta w_{ij} + |w_{ij}| \quad (9.16)$$

Существуют и другие типы методов регуляризации, такие как прямое изменение архитектуры сети наряду с фактическим обучением сети — мы рассмотрим эти методы в следующих главах.

С точки зрения того, какой метод регуляризации вы должны использовать (включая отсутствие в обще), вы должны рассматривать этот выбор как гиперпараметр, который вам необх одимо оптимизировать, и проводить эксперименты чтобы определить, следует ли применять регуляризацию, и если да, то какой метод регуляризации, и каково правильное значение λ . Дополнительные сведения о регуляризации см. в главе 7 работы Goodfellow et al. [10], раздел «Регуляризация» из учебника DeepLearning.net [99] и примечания к лекции Karpathy cs231n Neural Networks II [100].

9.4.4 Применение регуляризации к классификации изображений

Чтобы продемонстрировать регуляризацию в действии, давайте напишем код на Python, чтобы применить его к нашему набору данных «Животные». Откройте новый файл, назовите его Regularization.py и вставьте следующий код:

```
1 # импортируйте необх одимые пакеты
2 из sklearn.linear_model импортируйте SGDClassifier 3 из
sklearn.preprocessing import LabelEncoder 4 из
sklearn.model_selection import train_test_split 5 из
pyimagesearch.preprocessing import SimplePreprocessor 6 из
pyimagesearch.datasets import SimpleDatasetLoader 7 из imutils import paths
8 import argparse
```

Строки 2-8 импортируют необх одимые пакеты Python. Мы уже видели все эти импорты за исключением scikit-learn SGDClassifier. Как следует из названия этого класса, эта реализация инкапсулирует все концепции, которые мы рассмотрели в этой главе, в том числе:

- Функции потерь •

Количество эпох •

Скорость обучения •

Условия регуляризации

Таким образом, это идеальный пример для демонстрации всех этих концепций в действии.

Затем мы можем проанализировать аргументы командной строки и получить список изображений с диска:

```
10 # построить разбор аргумента и разобрать аргументы 11 ap =
argparse.ArgumentParser() 12 ap.add_argument("-d", "--dataset",
required=True,
13         help="путь к вх одному набору
данью ") 14 args = vars(ap.parse_args())
15
16 # получаем список путей к
изображениям 17 print("[INFO] загрузка
изображений...") 18 imagePaths = list(paths.list_images(args["dataset"]))
```

Учитывая пути к изображениям, мы изменим их размер до 32x32 пикселей, загрузим с диска в память, а затем свести их в массив из 3072 тусклых элементов:

```
20 # инициализировать преобразователь изображения загрузить набор данных с
диска, 21 # и изменить форму матрицы данных 22 sp = SimplePreprocessor(32, 32)
23 sdl = SimpleDatasetLoader(preprocessors=[sp]) 24 (data, labels) = sdl.load (imagePaths,
verbose=500) 25 data = data.reshape((data.shape[0], 3072))
```

Мы также закодируем метки как целевые числа и выполним разделение обучающей тестирования используя 75% данные для обучения и остальные 25% для тестирования

```

27 # кодировать метки как целевые числа
28 файл = LabelEncoder()
29 метки = le.fit_transform(метки)
30
31 # разделить данные на тренировочную и тестовую части, используя 75%
32 # данные для обучения и остальные 25% для тестирования
33 (trainX, testX, trainY, testY) = train_test_split(данные, метки,
34                                         test_size=0.25, random_state=5)

```

Давайте применим несколько различных типов регуляризации при обучении нашего SGDClassifier:

```

36 # цикл по нашему набору регуляризаторов
37 для яг в (Нет, "l1", "l2"):
38     # обучить классификатор SGD, используя функцию потерь softmax и
39     # заданная функция регуляризации для 10 эпох
40     print("Обучая модель [INFO] со штрафом {}".format(r))
41     модель = SGDClassifier(потеря="журнал", штраф=r, max_iter=10,
42                            Learning_rate="константа", eta0=0.01, random_state=42)
43     модель.fit(поездХ, поездY)
44
45     # оцениваем классификатор
46     акк = модель.оценка(тестХ, тестY)
47     print("[INFO] {} точность штрафа: {:.2f}%.format(r,
48           согл * 100))

```

Строка 37 повторяет наши регуляризаторы в том числе без регуляризации. Затем мы инициализируем обучаем SGDClassifier в строках 41-43.

Мы будем использовать кросс-энтропийную потерю со штрафом за регуляризацию r и λ по умолчанию, равным 0,0001. Мы будем использовать SGD для обучения модели в течение 10 эпох со скоростью обучения = 0,01. Затем мы оцениваем классификатор и отобразить результат точности на нашем экране в строках 46-48.

Чтобы увидеть нашу модель SGD, обученную различными типами регуляризации, просто выполните следующую строку команды:

```
$ pythonregularization.py --dataset ..datasets/animals
[INFO] загрузка изображений...
...
[INFO] Тренировочная модель со штрафом «Нет»
[INFO] Точность штрафа «Нет»: 50,40%
[INFO] Тренировочная модель со штрафом «l1»
[INFO] Точность штрафа «l1»: 52,53%
[INFO] Тренировочная модель со штрафом «l2»
[INFO] Точность штрафа «l2»: 55,07%
```

Мы видим, что без регуляризации мы получаем точность 50,40%. Использование регуляризации L1 наша точность увеличивается до 52,53%. Регуляризация L2 имеет наивысшую точность 55,07%.

 Использование разных значений random_state для train_test_split даст разные результаты. Набор данных здесь слишком мал, а классификатор слишком упрощен, чтобы увидеть полное влияние

регуляризация так что считайте это «рабочим примером». По мере того как мы продолжим работу над этой книгой, вы увидите более продвинутые варианты использования регуляризации, которые существенно повлияют на вашу точность.

На самом деле этот пример слишком мал, чтобы показать все преимущества применения регуляризации — для этого нам придется подождать, пока мы не начнем обучать сверточные нейронные сети. Однако в тоже время просто оцените, что регуляризация может повысить точность нашего тестирования и уменьшить переоснащение, если мы сможем правильно настроить гиперпараметры

9.5 Резюме

В этой главе мы раскрыли капот глубокого обучения и глубоко погрузились в механиズм, на котором работают современные нейронные сети — градиентный спуск. Мы исследовали два типа градиентного спуска:

1. Стандартный ванильный вкус.
2. Стохастическая версия, которая использует счастье.

Ванильный градиентный спуск выполняет только одно обновление веса за эпоху, что делает очень медленной (если не невозможной) сходимость на больших наборах данных. Стохастическая версия вместо этого применяет несколько обновлений веса за эпоху, вычисляя градиент на небольших мини-пакетах. Используя SGD, мы можем значительно сократить время сходимости обучения модели, а также получить меньшие потери и более высокую точность. Типичные размеры партий включают 32, 64, 128 и 256 штук.

Алгоритмы градиентного спуска контролируются скоростью обучения этого, безусловно, самое важное. Для правильной настройки при обучении собственных моделей.

Если ваша скорость обучения слишком велика, вы просто будете прыгать по ландшафту потерь и фактически не «изучите» какие-либо закономерности из своих данных. С другой стороны если ваша скорость обучения слишком мала, потребуется неизмеримо большее количество итераций, чтобы достичь даже разумных потерь. Чтобы сделать это правильно, вам нужно будет потратить большую часть своего времени на настройку скорости обучения.

Затем мы обсудили регуляризацию, которая определяется как «любой метод, повышающий точность тестирования в возможно, за счет точности обучения». Регуляризация включает в себя широкий спектр методов. Мы специально сосредоточились на методах регуляризации, которые применяются к нашим функциям потерь и правилам обновления веса, включая регуляризацию L1, регуляризацию L2 и Elastic Net.

Сточки зрения глубокого обучения нейронных сетей вы часто будете видеть регуляризацию L2, используемую для классификации изображений — хитрость заключается в настройке параметра λ , чтобы включить только нужное количество регуляризации.

На данный момент у нас есть прочная основа машинного обучения, но нам еще предстоит исследовать нейронные сети или обучить пользовательскую нейронную сеть с нуля. Всё это изменится в нашей следующей главе, где мы обсудим нейронные сети, алгоритм обратного распространения ошибки и способы обучения ваших собственных нейронных сетей на пользовательских наборах данных.

10. Основы нейронной сети

В этой главе мы подробно изучим основы нейронных сетей. Мы начнем с обсуждения искусственных нейронных сетей и того, как они вдохновлены реальными биологическими нейронными сетями в наших собственных тела. Оттуда мы рассмотрим классический алгоритм Perceptron и роль, которую он сыграл в истории нейронных сетей.

Основываясь на Perceptron, мы также изучим алгоритм обратного распространения краевого камня современного нейронного обучения — без обратного распространения мы не смогли бы эффективно обучать наши сети. Мы также реализуем обратное распространение с помощью Python с нуля, чтобы убедиться, что мы понимаем этот важный алгоритм.

Конечно, современные библиотеки нейронных сетей, такие как Keras, уже имеют (в высоком изложении) встроенные алгоритмы обратного распространения ошибки. Реализация обратного распространения вручную каждый раз, когда мы хотим обучить нейронную сеть, была бы похожа на кодирование структуры данных связанных списков или хештаблиц с нулем каждый раз, когда мы работали над задачей программирования общего назначения — это не только не реально, но и пустая тратить время. наше время ресурсы. Чтобы упростить процесс, я покажу, как создавать стандартные нейронные сети с прямой связью, используя библиотеку Keras.

Наконец, мы завершим эту главу обсуждением четырех ингредиентов, которые вам понадобятся, когда вы построите любую нейронную сеть.

10.1 Основы нейронной сети

Прежде чем мы можем работать со сверточными нейронными сетями, нам сначала нужно понять основы нейронных сетей. В этом разделе мы рассмотрим:

- Искусственные нейронные сети и их связь с биологией.
- Оригинальный алгоритм Perceptron.
- Алгоритм обратного распространения как его можно использовать для обучения многослойных нейронных сетей эффективно.
- Как обучать нейронные сети с помощью библиотеки Keras.

К тому времени, когда вы законите эту главу, вы будете хорошо понимать нейронные сети и сможете перейти к более продвинутым сверточным нейронным сетям.

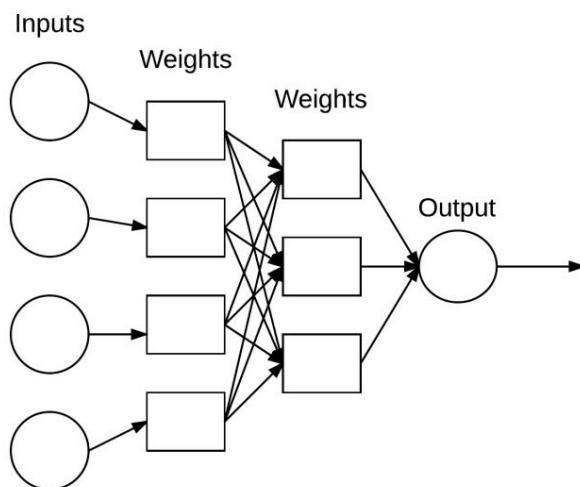


Рисунок 10.1: Простая архитектура нейронной сети. В ходе представлены сеть. Каждое соединение передает сигнал через два скрытых слоя сети. Последняя функция вычисляет метку выхода одного класса.

10.1.1 Введение в нейронные сети

Нейронные сети являются строительными блоками систем глубокого обучения. Чтобы добиться успеха в глубоком обучении, нам нужно начать с изучения основ нейронных сетей, включая архитектуру, типы узлов и алгоритмы «обучения» наших сетей.

В этом разделе мы начнем с общего обзора нейронных сетей и стоящих за ними мотивов, включая их отношение к биологии человеческого разума. Оттуда мы обсудим наиболее распространенный тип архитектуры нейронные сети с прямой связью. Мы также кратко обсудим концепцию нейронного обучения то, как оно позже будет связано с алгоритмами, которые мы используем для обучения нейронных сетей.

Что такое нейронные сети?

Многие задачи, связанные с интеллектом, распознаванием образов и обнаружением объектов, чрезвычайно трудно автоматизировать, но, похоже, они легко естественно выполняются живыми и маленькими детьми. Например, как ваша домашняя собака узнает вас, владельца, по сравнению с совершенно незнакомым человеком? Как маленькому ребенку научиться распознавать разницу между школьным автобусом и маршрутным автобусом? И как наш собственный мозг подсознательно выполняет сложные задачи по распознаванию образов каждый день, даже не замечая этого?

Ответ лежит в наших собственных телах. Каждый из нас содержит реальную биологическую нейронную сеть, которая связана с нашей нервной системой — эта сеть состоит из большого количества взаимосвязанных нейронов (нейронных клеток).

Слово «нейронный» является формой прилагательного от «нейрон», а «сеть» обозначает графоподобную структуру; следовательно, «искусственная нейронная сеть» — это вычислительная система, которая пытается имитировать (или, по крайней мере, вдохновлена) нейронные связи в нашей нервной системе. Искусственные нейронные сети также называют «нейронными сетями» или «искусственными нейронными системами». Искусственные нейронные сети принятосокращать и называть их «ANN» или просто «NN» — я буду использовать обе аббревиатуры в остальной части книги.

Чтобы система считалась NN, она должна содержать помеченную структуру ориентированного графа, в которой каждый узел графа выполняет некоторые простые вычисления. Из теории графов мы знаем, что ориентированный граф состоит из набора узлов (т. е. вершин) и набора соединений (т. е. ребер), соединяющих вместе пары узлов. На рис. 10.1 мы можем видеть пример такого графа NN.

Каждый узел выполняет простое вычисление. Затем каждое соединение переносит сигнал (т. е. результат вычислений) от одного узла к другому, помеченный весом коэффициентом, указывающим степень усиления или ослабления сигнала. Некоторые соединения имеют большие положительные веса, которые усиливают сигнал, указывая на то, что сигнал очень важен при классификации. Другие имеют отрицательные веса, уменьшающие силу сигнала, таким образом указывая, что вклад узла менее важен в окончательной классификации. Мы называем такую систему искусственной нейронной сетью, если она состоит из графовой структуры (как на рис. 10.1) с весами соединений, которые можно изменить с помощью алгоритма обучения.

Онение к биологии

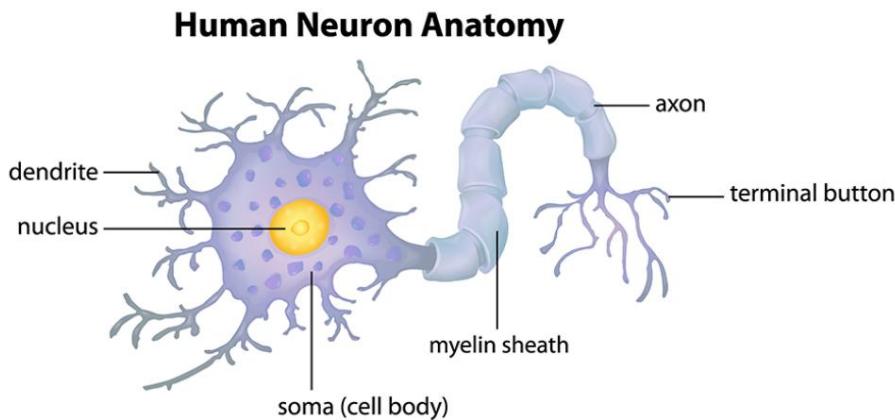


Рисунок 10.2: Структура биологического нейрона. Нейроны связываются друг с другом через их дендриты и аксоны.

Наш мозг состоит примерно из 10 миллиардов нейронов, каждый из которых связан примерно с 10 000 других нейронов. Тело клетки нейрона называется soma, где вхолода (дендриты) и выхода (аксоны) соединяются с другой сомой (рис. 10.2).

Каждый нейрон получает электрические сигналы от других нейронов на своих дендритах. Если эти электрические входы достаточно сильны, чтобы активировать нейрон, то он активирует другой нейрон, передавая его подендритам других нейронов. Эти прикрепленные нейроны также могут возбуждаться продолжая таким образом, процесс передачи сообщения.

Ключевым введением здесь является то, что срабатывание нейрона — это бинарная операция: нейрон либо срабатывает, либо не срабатывает. Нет разницы «степени» стрельбы. Проще говоря, нейрон срабатывает только в том случае, если общий сигнал, полученный в соме, превышает заданный порог.

Однако имейте в виду, что ИИС просто вдохновлены тем, что мы знаем о мозге и о том, как он работает. Цель глубокого обучения — в том, чтобы имитировать работу нашего мозга, а в том, чтобы брать фрагменты, которые мы понимаем, и позволять нам проводить аналогичные параллели в нашей собственной работе. В конце концов, мы недостаточно знаем о нейронах и более глубоких функциях мозга, чтобы правильно моделировать его работу — вместо этого мы черпаем вдохновение и движемся дальше.

Искусственные модели

Давайте начнем с рассмотрения базовой ИС, выполняющей простое вычисление суммирования всех одних данных, как показано на рис. 10.3. Значения x_1 , x_2 и x_3 являются одними данными для нашей NN и обычно соответствуют одной строке (то есть точке данных) из нашей матрицы проекта. Постоянное значение 1 — это наше смещение, которое, как предполагается, встроено в матрицу проекта. Мы можем думать об этих векторах данных как о векторах признаков для NN.

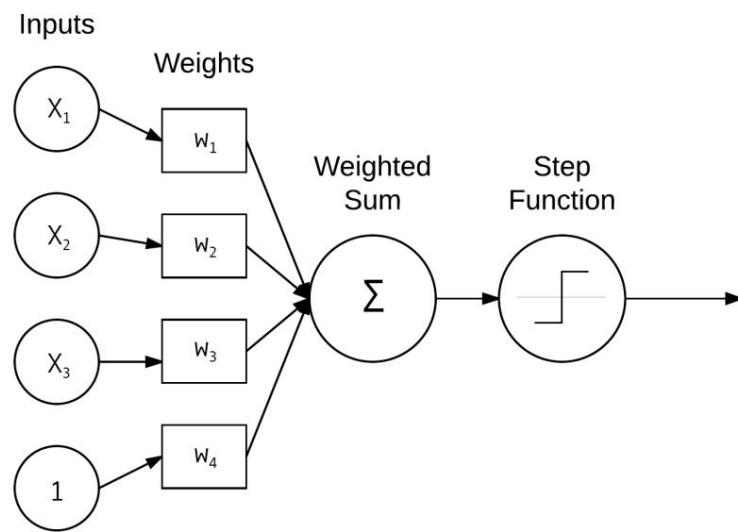


Рисунок 10.3: Простая NN, которая принимает взвешенную сумму вх однък данък х и весов w. Затем эта взвешенная сумма передается через функцию активации, чтобы определить, срабатывает ли нейрон.

На практике эти вх однъе данные могут быть векторами, используемыми для количественной оценки содержимого изображения систематическим, предопределенным образом (например, цветовые гистограммы гистограмма ориентированых градиентов [32], локальные бинарные шаблоны [21] и т. д.). В контексте глубокого обучения эти вх однъе данные представляют собой необработаные интенсивности пикселей самих изображений.

Каждый х связывается с нейроном через весовой вектор W , состоящий из w_1, w_2, \dots, w_n , что означает, что для каждого x_i в x есть связанный вес w_i .

Наконец, вх однъе узел справа на рис. 10.3 берет взвешенную сумму, применяет функцию активации f (используемую для определения «срабатывает» ли нейрон) и вводит значение. В результате математически, вы обычно сталкиваетесь со следующими тремя формами:

$$f(w_1x_1 + w_2x_2 + \dots + w_nx_n) = f(\sum_{i=1}^n w_i x_i) \quad \text{Или просто } f(\text{net}), \text{ где net} = \sum_{i=1}^n w_i x_i$$

сумма вх однък данък с последующим применением функции активации f .

Функции активации

Самой простой функцией активации является «ступенчатая функция», используемая алгоритмом персептрона (которую мы рассмотрим в следующем разделе).

$$f(\text{net}) = \begin{cases} 0 & \text{иначе} \\ 1 & \text{если } \text{net} > 0 \end{cases}$$

Как видно из приведенного выше уравнения, это очень простая пороговая функция. Если взвешенная сумма net больше или равна нулю, то $f(\text{net})$ по оси y , мы можем понять, почему эта функция активации получила свое название (рис. 10.4, вверху слева). Вх од f всегда равен нулю, когда net меньше или равно нулю. Если net больше нуля, то f вернет единицу. Таким образом, эта функция выглядит как ступенька лестницы, мало чем отличающаяся от лестницы, по которой вспоминается и спускается каждый день.

Однако, будучи интуитивно понятной и простой в использовании, ступенчатая функция не дифференцируема, что может привести к проблемам при применении градиентного спуска и обучения нашей сети.

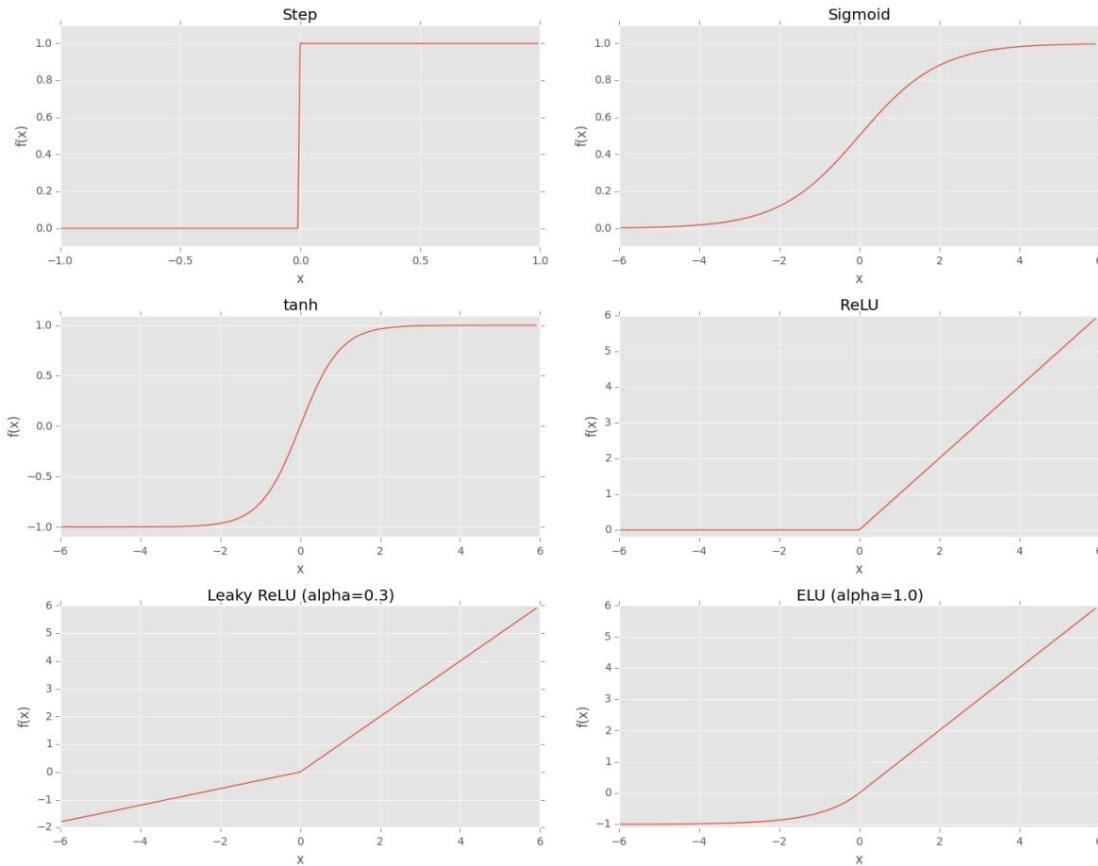


Рисунок 10.4: Вверху слева: Ступенчатая функция. Вверху справа: сигмовидная функция активации. В середине слева: гиперболический тангенс. В середине справа: активация ReLU (наиболее часто используемая функция активации для глубоких нейронных сетей). Внизу слева: Leaky ReLU, варант ReLU, допускающий отрицательные значения. Внизу справа: ELU, еще один вариант ReLU, который часто работает лучше, чем Leaky ReLU.

Вместо этого более распространенной функцией активации, используемой в истории литературы по нейронным сетям, является сигмовидная функция (рис. 10.4, вверху справа), которая следует уравнению:

$$t = \frac{1}{1 + e^{-x}} \quad (10.1)$$

Сигмовидная функция лучше подходит для обучения, чем простая ступенчатая функция, поскольку она:

1. Везде непрерывна и дифференцируема.

2. Симметрична относительно оси Y.

3. Асимптотически приближается к своим значениям насыщения.

Основным преимуществом здесь является то, что гладкость сигмовидной функции упрощает разработать алгоритмы обучения. Однако с сигмовидной функцией есть две большие проблемы:

1. Весь одногипотетичный центрированы нулю.

2. Насыщенные нейроны существуют градиент, так как дельта градиента будет чрезвычайно небольшой.

Гиперболический тангенс, или тангенс (с пах ожея формой сигмовидной), также активно использовался в качестве функции активации вплоть до конца 1990-х годов (рис. 10.4, в середине слева): уравнение для тангенса выглядит следующим образом:

$$f(z) = \text{th}(z) = (e^z - e^{-z}) / (e^z + e^{-z}) \quad (10.2)$$

Функция \tanh центрирована по нулю, но градиенты по-прежнему исчезают, когда нейроны насыщаются.

Теперь мы знаем, что есть лучший выбор для функций активации, чем сигмоидальная тангенсная функции. В частности, работа Hahnloser et al. в своей статье 2000 года «Цифровая селекция аналоговых усиления существуют в кремниевой схеме, вдохновленной кортексом» [101], представили в прямую линейную единицу (ReLU), определяемую как:

$$e(x) = \max(0, x) \quad (10.3)$$

ReLU также называют «линейными функциями» из-за того, как они выглядят на графике (рис. 10.4, в середине справа). Обратите внимание, как функция равна нулю для отрицательных x одних данных, а затем линейно увеличивается для положительных значений. Функция ReLU не насыщается, а также через вынужденный эффективен в вычислительном отношении.

Эмпирически функция активации ReLU имеет тенденцию превосходить как сигмоидную, так и тангенциальную функции почти во всех приложениях. В сочетании с работой Ханлозера и Сунга в их последнюю статью 2003 г. «Разрешенные и запрещенные множества в симметричных пороговых линейных сетях» [102] было обнаружено, что функция активации ReLU имеет более сильные биологические мотивы, чем предыдущие семейства функций активации, в том числе более полное математическое обоснование.

По состоянию на 2015 год ReLU является самой популярной функцией активации, используемой в глубоком обучении [9]. Однако, возникает проблема, когда у нас значение равно нулю — градиент нельзя взять.

Вариант ReLU, называемый Leaky ReLU [103], допускает небольшой ненулевой градиент, когда устройство активно:

$$\begin{aligned} & \text{если } \text{нетто} \geq 0 \\ f(\text{нетто}) &= (\text{нетто})^\alpha \quad \text{иначе} \end{aligned}$$

Построив эту функцию на рисунке 10.4 (внизу слева), мы видим, что функция действительна разрешена. принимать отрицательное значение, в отличие от традиционных ReLU, которые «зажимают» ввод функции на нуле.

Параметрические ReLU, или сокращенно PReLU [96], основанные на Leaky ReLU и позволяют изучать параметр α на основе активации за активацией, подразумевая, что каждый узел в сети может изучать другой «коэффициент утечки» отдельно от остальных узлов.

Наконец, у нас также есть экспоненциальные линейные единицы (ELU), введенные Clevert et al. в их статью 2015 г. Быстрое и точное глубокое обучение с помощью экспоненциальных линейных единиц (ELU) [104]:

$$\begin{aligned} & \text{если } \text{нетто} \geq 0 \\ f(\text{нетто}) &= (\text{нетто})^\alpha (\exp(\text{нетто}) - 1) \quad \text{в противном случае} \end{aligned}$$

Значение α является постоянным и устанавливается при создании экземпляра сетевой архитектуры — это не пах оже на PReLU, где изучается. Типичное значение для $\alpha = 1.0$. Рисунок 10.4 (внизу справа) визуализирует функцию активации ELU.

Благодаря работе Clevert et al. (и мои собственные анекдотические эксперименты), ELU часто обеспечивают более высокую точность классификации, чем ReLU. ELU редко, если вообще когда-либо, работают хуже, чем ваша стандартная функция ReLU.

Какую функцию активации я использую?

Учитывая популярность самого последнего в оплощни губокого обучения, произошел связанный с этим взрыв функций активации. Из-за большого количества вариантов функций активации, как современных (ReLU, Leaky ReLU, ELU и т. д.), так и «классических» (ступенчатая сигмовидная тангенциальная и т. д.), это может показаться пугающим, возможно, даже подавляющим. задача выбрать соответствующую функцию активации.

Однако почти во всех ситуациях я рекомендую начинать с ReLU, чтобы получить базовую точность (как и в большинстве статей, опубликованных в литературе по глубокому обучению). Оттуда вы можете попробовать заменить свой стандартный ReLU на вариант Leaky ReLU.

Лично я предпочитаю начинать с ReLU, настраивать параметры сети и оптимизатора (архитектура, скорость обучения, сила регуляризации и т. д.) и обращать внимание на точность. Как только я удовлетворен точностью, я заменяю ELU и часто замечаю улучшение точности классификации на 1-5% в зависимости от набора данных. Отметь же, этот только мой anecdотический совет. Вы должны провести свои собственные эксперименты, отметить свои результаты и, как правило, начните с обычного ReLU и настройте другие параметры в вашей сети, а затем замените некоторые из более «экзотических» вариантов ReLU.

Сетевые архитектуры с прямой связью

Хотя существует множество различных архитектур NN, наиболее распространенной архитектурой является сеть с прямой связью, как показано на рис. 10.5.

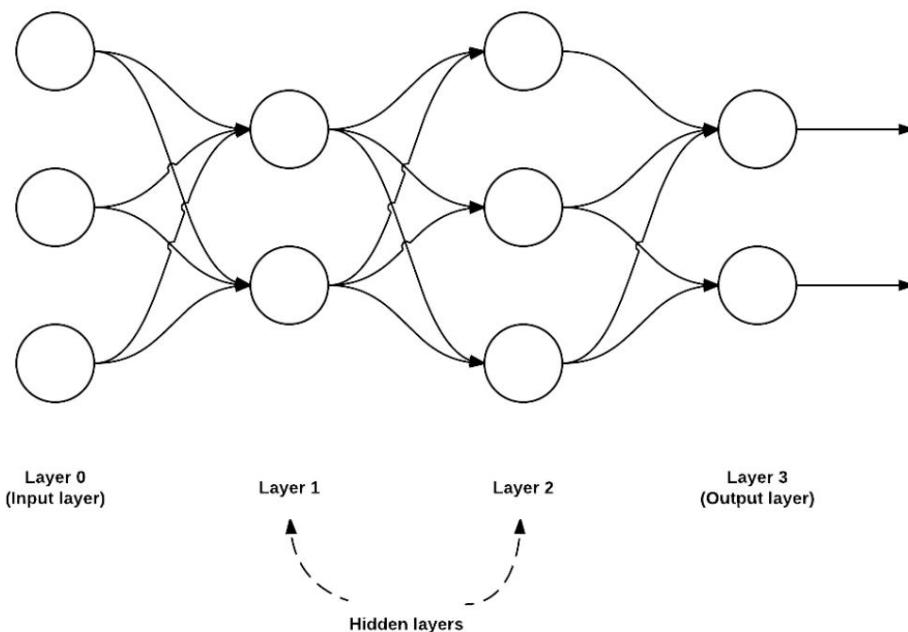


Рисунок 10.5: Пример нейронной сети с прямой связью с 3 вх однми узлами, скрытым слоем с 2 узлами, вторым скрытым слоем с 3 узлами и конечным вх однм слоем с 2 узлами.

В архитектуре этого типа соединение между узлами разрешено только от узлов уровня i к узлам уровня $i+1$ (отсюда и термин «прямая связь»). Обратные или межуровневые соединения запрещены. Когда сети с прямой связью включают соединения с обратной связью (вх одные соединения, которые возвращаются вх оды), они называются рекуррентными нейронными сетями.

В этой книге мы сосредоточимся на нейронных сетях с прямой связью, поскольку они являются краеугольным камнем современного глубокого обучения, применяемого к компьютерному зрению. Как мы узнаем в главе 11, сверточные нейронные сети — это просто частный случай нейронной сети с прямой связью.

Чтобы описать сеть с прямой связью, мы будем использовать последовательность цепочек чисел для быстрого и краткого ранения количества узлов в каждом слое. Например, сеть на рис. 10.5 выше представляет собой сеть с прямой связью 3-2-3-2:

Слой 0 содержит 3 вектора, наши значениях. Это может быть необработанная интенсивность пикселей изображения или вектор признаков, извлеченный из изображения.

Слой 1 и 2 являются скрытыми слоями, содержащими 2 и 3 узла соответственно.

Уровень 3 — это вектор одной слой или видимый слой — именно здесь мы получаем общую классификацию вектора из нашей сети. Вектор одной слой обычно имеет столько узлов, сколько меток классов; один узел для каждого потенциального вектора. Например, если бы мы построили NN для классификации рукописных цифр, наш вектор одной слой состоял бы из 10 узлов, по одному для каждой цифры от 0 до 9.

Нейронное обучение

Нейронное обучение относится к методу изменения весов и связей между узлами в сети. Биологически мы определяем обучение с точки зрения принципа Хебба:

«Когда аксон клетки A начинает достаточно близко, чтобы взвесить клетку B, и однократно или постоянно возбуждает ее, в одной или обеих клетках происходит однократный процесс роста или метаболические изменения так, что эффективность A, как одной из клеток, возбуждающих B, увеличивается» — Дональд Хебб [105]

С точки зрения ИИС этот принцип подразумевает, что должна увеличиваться связь между узлами, имеющими аналогичные вектора одни данные, при представлении одинаковых векторов одних данных. Мы называем это корреляционным обучением, потому что связь между нейронами в конечном итоге представляет собой корреляцию между векторами одних данных.

Для чего используются нейронные сети?

Нейронные сети можно использовать как в контролируемых, так и в неконтролируемых и частично контролируемых учебных задачах, конечно, при условии использования соответствующей архитектуры. Полный обзор ИС включает в себя рамки этой книги (см. подробный обзор глубоких искусственных сетей у Шилдхубера [40] и обзор классических методов у Мехротры [106]); однако общие применения NN включают классификацию, регрессию, кластеризацию, векторное квантование, ассоциацию шаблонов и аппроксимацию функций, и это лишь некоторые из них.

На самом деле почти для каждого аспекта машинного обучения ИС применялись в той или иной форме. В контексте этой книги мы будем использовать NN для компьютерного зрения и классификации изображений.

Краткое изложение основ нейронных сетей

На этом разделе мы рассмотрели основы искусственных нейронных сетей (ИС или просто ИС).

Мы начали с изучения биологической мотивации ИС, а затем узнали, как можно математически определить функцию, имитирующую активацию нейрона (то есть функцию активации).

Основываясь на этой модели нейрона, мы можем определить архитектуру сети, состоящей из (как минимум) вектора одного слоя и вектора одного слоя. Некоторые сетевые архитектуры могут включать несколько скрытых слоев между векторами одних и тех же данных слоями. Наконец, каждый слой может иметь один или несколько узлов. Узлы в одном слое не содержат функции активации (они «где» в однородности и генерируют пиксели нашего изображения); однако узлы как в скрытом, так и в векторном слоях содержат функцию активации.

Мы также рассмотрели три популярные функции активации: сигмоидную, тангенциальную и ReLU (и ее варианты).

Традиционно для обучения сетей использовались сигмоиды и тангенциальные функции; Однако, после статьи Ханновера и др. 2000 г. [101] функция ReLU стала использоваться чаще.

В 2015 году ReLU стала самой популярной функцией активации, используемой в архитектурах глубокого обучения [9]. Основываясь на успехе ReLU, у нас также есть Leaky ReLU, вариант ReLU, который стремится

улучшить производительность сети, позволив функции принимать отрицательное значение. Семейство функций Leaky ReLU состоит из стандартного варианта ReLU с утечкой, PReLU и ELU.

Наконец, важно отметить, что отямы фокусируемся на глубоком обучении строгом контексте классификации изображений, нейронные сети в той или иной степени использовались почти во всех нишах машинного обучения.

Теперь, когда мы познакомились с основами NN, давайте сделаем это знание более конкретным, изучив реальную архитектуру и связывание с ними реализации. В следующем разделе мы обсудим классический алгоритм Perceptron, один из первых когда-либо созданных ИНС.

10.1.2 Алгоритм персептрона В первые

представленный Розенблаттом в 1958 году, Персептрон в вероятностная модель хранения информации в мозгу [12] является возможно, самыми старыми и самыми простыми из алгоритмов ИНС. После этой публикации методы основанные на персептроне, были в ярости в сообществе нейронных сетей. Одна только эта статья имела огромной степени ответственности за популярность и полезность нейронных сетей сегодня.

Но затем, в 1969 году, в сообществе машинного обучения обрушилась «Зима ИИ», которая почти навсегда заморозила нейронные сети. Мински и Пейперт опубликовали «Персептроны введение в вычислительную геометрию» [14], книгу, которая почти на десятилетие практически затормозила исследования в области нейронных сетей. Perceptron не может разделить нелинейные точки данных.

Учитывая что большинство реальных наборов данных естественным образом нелинейно разделимы, казалось, что Perceptron, наряду с остальными исследованиями в области нейронных сетей, может закончиться безвременно.

Между публикацией Мински и Пейперта и невыполнимыми обещаниями нейронных сетей, революционизирующими промышленность, интерес к нейронным сетям существенно снизился. Только когда мы начали исследовать более глубокие сети (иногда называемые многослойными персепtronами) вместе с алгоритмом обратного распространения ошибки (Вербос [15] и Румельхарт [16]), «зима ИИ» в 1970-х закончилась и начались исследования нейронных сетей. снова нагреть.

Тем не менее, персептрон по-прежнему является очень важным алгоритмом для понимания, поскольку он готовит почву для более продвинутых многоуровневых сетей. Мы начнем этот раздел с обзора архитектуры персептрона и объясним произведенное обучение (называемое делта-правилом), используемое для обучения персептрона. Мы также рассмотрим критерии прекращения работы сети (т. е. когда персептрон должен прекратить обучение). Наконец, мы реализуем алгоритм Perceptron на чистом Python и используем его для изучения изучения этого, почему сеть не может изучать нелинейно разделяемые наборы данных.

Наборы данных AND, OR и XOR

Прежде чем мы приступим к изучению самого персептрона, давайте сначала обсудим «побитовые операции», включая AND, OR и XOR (исключающее ИЛИ). Если вы уже прошли курс информатики начального уровня, возможно, вы уже знакомы с побитовыми функциями.

Побитовые операторы связывания с ними побитовые наборы данных принимают два вх однократных бита и создают окончательный вх однократного бита после применения операции. Имея два вх однократных бита, каждый из которых потенциально может принимать значение 0 или 1, существует четыре возможных комбинации этих двух битов. В таблице 10.1 представлены возможные вх однократные и вх однократные значения для AND, ИЛИ и XOR:

Как мы видим слева, логическое И истинно тогда и только тогда, когда оба вх однократных значения равны 1. Если любое из вх однократных значений равно 0, И возвращает 0. Таким образом, существует только одна комбинация $x_0 = 1$ и $x_1 = 1$, когда вх однократных AND истинен.

В середине у нас есть операция ИЛИ, которая верна, когда только один из вх однократных значений равно 1. Таким образом, есть три возможных комбинации двух битов x_0 и x_1 , которые дают значение $y = 1$.

Наконец, справа отображается операция XOR, которая истинна тогда и только тогда, когда один из вх одновременно оба. В то время как у ИЛИ было три возможных ситуации, когда $y = 1$, у XOR есть только две.

x_0	x_1	$x_0 \& x_1$	x_0	x_1	$x_0 x_1$	x_0	x_1	$x_0 \oplus x_1$
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Таблица 10.1: Слева: набор данных побитового И. Учитывая два входа, выход равен 1, если оба входа равны 1. Средний: набор данных побитового ИЛИ. При наличии двух входов один из данных выход равен 1, если любой из двух входов один из данных равен 1. Справа: набор данных XOR (Exclusive OR). Учитывая два входа, выход равен 1 тогда и только тогда, когда один из входов равен 1, но не оба.

Мы часто используем эти простые «побитовые наборы данных» для тестирования и отладки алгоритмов машинного обучения. Если мы нарисуем визуализации значений AND, OR и XOR (где красные кружки обозначают нулевые результаты, а синие звездочки — единичные) на рис. 10.6, вы заметите интересную закономерность:

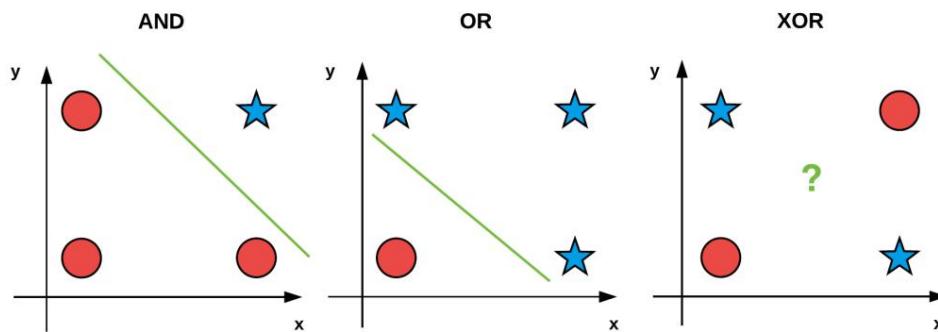


Рисунок 10.6. Наборы данных побитового И и ИЛИ линейно разделимы, что означает, что мы можем нарисовать одну линию (зеленую), которая разделяет два класса. Однако для XOR невозможно провести единую линию, разделяющую два класса, поэтому это нелинейно разделимый набор данных.

И И, и ИЛИ линейно разделимы — мы можем четко провести линию, разделяющую классы 0 и 1 — тоже самое не верно для XOR. Потратьте время, чтобы убедиться в том, что невозможно провести линию, четко разделяющую два класса в задаче XOR. Таким образом, XOR является примером нелинейно разделяемого набора данных.

В идеале мы хотели бы, чтобы наши алгоритмы машинного обучения могли разделять нелинейные классы, поскольку большинство наборов данных, встречающихся в реальном мире, нелинейны. Следовательно, при построении, отладке и оценке данного алгоритма машинного обучения мы можем использовать побитовые значения x_0 и x_1 в качестве нашей проектной матрицы и затем попытаться предсказать соответствующие значения.

В отличие от нашей стандартной процедуры разделения наших данных на обучающие и тестовые разбиения при использовании побитовых наборов данных мы просто обучаем и оцениваем нашу сеть на одном и том же наборе данных. Наша цель здесь — просто определить, может ли наш алгоритм обучения вообще изучить закономерности в данных. Как мы видим, алгоритм Perceptron может правильно классифицировать функции AND и OR, но не может классифицировать данные XOR.

Архитектура персептрона

Розенблатт [12] определил персептрон как систему, которая обучается используя помеченные примеры (т. е. контролируемое обучение) векторов признаков (или необработанных интенсивностей пикселей), сопоставляя эти входы одному из классов.

В своей простейшей форме персептрон содержит N входных узлов, по одному на каждую запись в векторах одной строке таблицы

матрица проекта, за которой следует только один уровень в сети только с одним узлом на этом уровне (рис. 10.7).

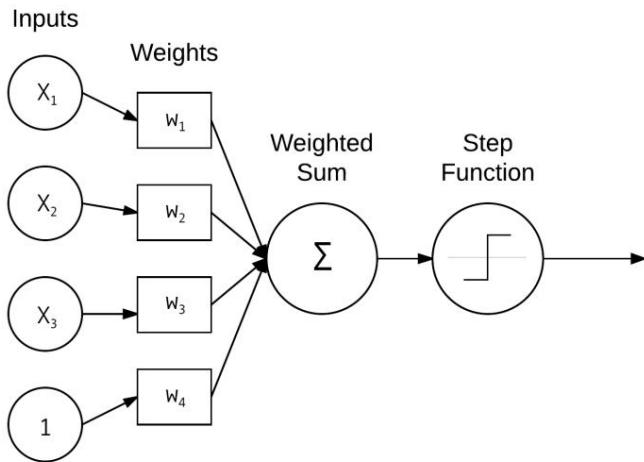


Рисунок 10.7: Архитектура сети Perceptron.

Существуют связи и соответствующие им веса w_1, w_2, \dots, w_i от входов одних x_i к единственному выходу одного узла в сети. Этот узел принимает взвешенную сумму всех одних данных и применяет ступенчатую функцию для определения метки выхода одного класса. Персептрон выдаст либо 0, либо 1 – 0 для класса №1 и 1 для класса №2; таким образом, в своей первоначальной форме персептрон представляет собой простобинарный двухклассовый классификатор.

Процедура обучения персептрана и дельта-правило

Обучение персептрана – довольно простая операция. Наша цель – получить набор весов w , который точно классифицирует каждый экземпляр в нашем обучающем наборе. Чтобы обучить наш персептрон, мы многократно передаем в сеть наши обучающие данные. Каждый раз, когда сеть видит полный набор обучающих данных, мы говорим, что это эпоха прошла. Обычно требуется много эпох, пока весовой вектор w можно будет узнать, чтобы они могли разделить наши два класса данных.

Псевдокод для алгоритма обучения Perceptron можно найти ниже: Фактическое «обучение» происходит в шагах 2b и 2c. Впервые, мы пропускаем вектор признаков x_j через сеть, взвешенное скалярное произведение весов w и получаем результат y_j . Затем это значение передается через ступенчатую функцию, которая возвращает 1, если $x > 0$, и 0 в противном случае.

Теперь нам нужно обновить наш весовой вектор w , чтобы сделать шаг в направлении, которое «ближе» к правильной классификации. Это обновление вектора веса обрабатывается дельта-правилом на шаге 2c.

Выражение $(d_j - y_j)$ определяет, правильна ли классификация в одних данных. Если классификация верна, то эта разница будет равна нулю. В противном случае разница будет либо положительной, либо отрицательной, указывая направление, в котором будут обновляться наши веса (в конечном итоге приближаясь к правильной классификации). Затем мы умножаем $(d_j - y_j)$ на x_j , приближаясь к правильному классификатору.

1. Инициализировать наш весовой вектор w небольшими случайными значениями
2. Пока персептрон не сойдется

- (a) Перебрать каждый вектор признаков x_j и метку истинного класса d_j в нашем обучающем наборе D
- (b) Взять x_j и пропустите его через сеть, вычислив в $y_j = f(w(t) \cdot x_j)$
- (c) Обновите веса w : $w_i(t+1) = w_i(t) + \eta(d_j - y_j)x_j, i$ для всех признаков $0 \leq i \leq n$

Рисунок 10.8: Процедура обучения алгоритму Perceptron.

Значение α является наивысшей скоростью обучения и определяет, насколько большим (или маленьким) шаг мы делаем. Очень важно, чтобы это значение было установлено правильно. Большое значение α заставит нас сделать шаг в правильном направлении; однако этот шаг может быть слишком большим, и мы можем легко выйти за локальный/глобальный оптимум.

И наоборот, небольшое значение α позволяет нам делать крохотные шаги в правильном направлении, гарантировая что мы не превысим локальный/глобальный минимум; тем не менее, эти крохотные детские шаги требуют непреодолимого количества времени, чтобы наше обучение сх одилось.

Наконец, мы добавляем к предыдущему весовому вектору в момент времени t $w_j(t)$, что завершает процесс «перехода» к правильной классификации. Если эта процедура обучения покажется вам немного запутанной, не беспокойтесь — мы подробно рассмотрим ее с помощью кода Python позже в Разделе 10.1.2.

Прекращение обучения персептрона

Процесс обучения персептрона может продолжаться до тех пор, пока все обучающие выборки не будут правильно классифицированы или не будет достигнуто заданное количество эпох. Завершение обеспечивается если а достаточном малом обучающему данное линейноразделимы

Итак, что произойдет, если наши данные не будут линейноразделимы или мы сделаем плохой выбор в α ? Обучение будет продолжаться бесконечно? В этом случае нет — мы обычно останавливаемся после достижения заданного количества эпох или если количество ошибочных классификаций не изменилось за большое количество эпох (указывая на то, что данные не являются линейноразделимыми). Для получения более подробной информации об алгоритме персептрона, пожалуйста, обратитесь либо к Стенфордской лекции Эндрю Ng [76], либо к в однъимглавам книги Mehrotra et al. [106].

Реализация персептрона на Python Теперь, когда мы изучили алгоритм персептрона, давайте реализуем сам алгоритм на Python. Создайте файл с именем `perceptron.py` в вашем пакете `pyimagesearch.nn` — в этом файле будет храниться наша реальная реализация `Perceptron`:

```
-- pyimagesearch | |---  
|__init__.py | |init_.py | |---  
perceptron.py
```

После того, как вы создали файл, откройте его и вставьте следующий код:

```
1 # импортируем необходимые пакеты  
2  
импортируем numpy как np  
3  
Персептрон4 класса:  
4  
5     def __init__(я N, альф а=0.1):  
6         # инициализируем матрицу весов и со временем скорость обучения  
7         self.W = np.random.randn(N + 1) / np.sqrt(N) self.alpha = alpha  
8
```

Строка 5 определяет конструктор для нашего класса `Perceptron`, который принимает единственное необходимое параметр, за которым следует второй необязательный:

1. N : количество столбцов в наших векторах признаков. В контексте наших побитовых наборов данных мы установим N равным двум, так как есть два вектора.
2. α : наша скорость обучения для алгоритма `Perceptron`. По умолчанию мы установим это значение равным 0,01.

Обычный выбор скорости обучения обычно находится в диапазоне $\alpha = 0,1, 0,01, 0,001$.

Строка 7 записывает нашу весовую матрицу W со случайными значениями, выбранными из «нормального» (гауссова) распределение с нулевым средним и единичной дисперсией. Весовая матрица будет иметь $N+1$ элементов, один для каждого из N в одних данных векторе признаков, плюс один для смещения. Делим W на квадратный корень количества в одних данных, распространенный метод, используемый для масштабирования нашей матрицы весов, что приводит к более быстрому сходимости. Мы рассмотрим методы инициализации весов позже в этой главе.

Далее определим ступенчатую функцию:

```
10     шаг определения(я x):
11         # применяем ступенчатую функцию
12         вернуть 1, если x > 0 иначе 0
```

Эта функция имитирует поведение ступенчатого уравнения в разделе 10.4 выше — если x положительное мы возвращаем 1, иначе мы возвращаем 0.

Чтобы на самом деле обучить персептрон, мы определим функцию с именем `fit`. Если у вас есть какие-либо предыдущие опыты работы с машинным обучением Python и библиотекой scikit-learn, тогда вы будете знать, что это общее название вашей процедуры обучения подходит для функции, например, «подгонка модели к данным»:

```
14     def fit(self, X, y, epochs=10):
15         # вставить столбец из 1 в качестве последней записи в функции
16         # матрица -- эта маленькая и строка позволяет нам лечить смещение
17         # как обучаемый параметр в весовой матрице
18         X = np.c_[X, np.ones((X.shape[0]))]
```

Метод подгонки требует двух параметров, за которыми следует один необязательный параметр:

Значение X — это наши фактические данные обучения. Переменная y — это метки нашего целевого класса (т.е. что наша сеть должна предсказывать). Наконец, мы поставляем эпохи и, количество эпох. Персептрон будет обучаться.

Строка 18 применяет трюк смещения (раздел 9.3), вставляя столбец единиц в обучающие данные, что позволяет нам рассматривать смещение как обучаемый параметр непосредственно внутри весовой матрицы.

Далее, давайте рассмотрим фактическую процедуру обучения

```
20     # перебираем желаемое количество эпох
21     для эпох в пр.arange(0, эпохи):
22         # цикл по каждой отдельной точке данных
23         для (x, цель) в zip(X, y):
24             # возводим скалярное произведение между x однами объектами
25             # и матрицу весов, затем передать это значение
26             # через ступенчатую функцию для получения предсказания
27             p = self.step(np.dot(x, self.W))
28
29             # выполнить обновление веса только в том случае, если наш прогноз
30             # не соответствует цели
31             если p != цель:
32                 # определяем ошибку
33                 ошибка = p - цель
34
35             # обновить матрицу весов
36             self.W += -self.alpha * ошибка * x
```

В строке 21 мы начинаем перебирать желаемое количество эпох. Для каждой эпохи мы также циклически перебираем над каждой отдельной точкой данных x и ввести метку целевого класса (строка 23).

Строка 27 берет скалярное произведение между вх одными функциями x и матрицей весов W , затем передает ввод через ступенчатую функцию для получения предсказания персептроном.

Применяют же процедуру обучения что и в листинге 10.8 выше, мы выполняем только взвешивание. обновить, если наш прогноз не соответствует цели (строка 31). Если это так, то определяем error (строка 33) путем вычисления знака (положительного или отрицательного) с помощью операции разности.

Обновление весовой матрицы выполняется строке 36, где мы делаем шаг к правильному классификации, масштабируя этот шаг по нашей скорости обучения альфа. В течение ряда эпох нашПерсептрон способен изучать закономерности в базовых данных и сдвигать значения весовой матрицы таким образом, что мыправильную классифицируем наши вх одные образцы.

Последняя функция которую нам нужно определить, — это прогнозирование, которое, как следует из названия используется для предсказать метки класса для заданного набора вх одных данных :

```

38     Def Predict (self, X, addBias=True):
39         # убедиться что наши вх одные данные являются матрицей
40         X = np.atleast_2d(X)
41
42         # проверяем, нужно ли добавлять столбец смещения
43         если добавить смещение:
44             # вставить столбец из 1 в качестве последней записи в функции
45             # матрица (смещение)
46             X = np.c_[X, np.ones((X.shape[0]))]
47
48             # возвьем скалярное произведение между вх одними объектами и
49             # весов ая матрица, затем передаем значение через шаг
50             # функция
51             вернуть self.step (np.dot (X, self.W))

```

Наш метод прогнозирования требует набора вх одных данных X , которые необх одимо классифицировать. Чек на линии 43 сделан чтобы видеть, нужно ли добавлять столбец смещения

Получение вх одных прогнозов для X такое же, как и процедура обучения — просто возвьмите скалярное произведение между вх одными функциями X и нашей матрицей весов W , а затем передать значение через наш ступенчатая функция Вывод шаговой функции возврашается вьющая функции.

Теперь, когда мыреализовали нашкласс Perceptron , давайте попробуем применить его к нашим побитовым наборам данных . и посмотрите, как работает нейронная сеть.

Оценка побитовых наборов данных персептрона

Дляначала давайте создадим файл с именем perceptron_or.py , который пыгается подогнать модель персептрона к побитове ИЛИ набор данных :

```

1 # импортируем необх одимые пакеты
2 из pyimagesearch.nn импортировать Perceptron
3 импортировать numpy как np
4
5 # построить набор данных ИЛИ
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7 y = np массив ([[0], [1], [1], [1]])
8
9 # определяем нашперсептрон и обучаем его
10 print("[INFO] обучаящийся персептрон...")
11 p = Персептрон(X.shape[1], альфа=0,1)
12 p.fit(X, y, эпох =20)

```

Строки 2 и 3 импортируют необходимые пакеты Python. Мы будем использовать нашу реализацию Perceptron из раздела 10.1.2 выше. Строки 6 и 7 определяют набор данных OR на основе таблицы 10.1.

Строки 11 и 12 обучают наш персептрон со скоростью обучения = 0,1 в общей сложности 20 эпох.

Затем мы можем оценить наш персептрон на данных, чтобы убедиться что он действительно изучил функцию ИЛИ:

14 # теперь, когда наш персептрон обучен, мы можем оценить его 15 print("[INFO] testing perceptron...")

```

16
17 # теперь, когда наша сеть обучена, выполните цикл по точкам данных
18 for (x, target) в zip(X, y): # сделайте прогноз для точки данных и
19     отобразите результат # на нашей консоли
20
21     pred = p.predict(x)
22     print("[INFO] data={}, наземная правда={}, пред={}".format(x,
23         target[0], pred))

```

В строке 18 мы перебираем каждую из точек данных в наборе данных ИЛИ. Для каждой из этих точек данных мы пропускаемых через сеть и получаем прогнозов (строка 21).

Наконец, строки 22 и 23 отображают точку в x одних данных, метку истинности, а также наш предсказал лейбл на нашу консоль.

Чтобы увидеть, способен ли наш алгоритм Perceptron изучить функцию ИЛИ, просто выполните следующую команду:

```
$ python perceptron_or.py
[INFO] обучая щий персептрон...
[INFO] тестирование персептрона...
[ИНФО] данные=[0 0], исх однайинф ормация=0,
предыдущая=0 [ИНФО] данные=[0 1], исх однайинф ормация=1,
предыдущая=1 [ИНФО] данные=[1 0], исх однайинф ормация
правда=1, пред=1 [ИНФО] данные=[1 1], основная правда=1, пред=1
```

Конечно, наша нейронная сеть способна правильно предсказать, что операция ИЛИ для $x_0 = 0$ и $x_1 = 0$ равна нулю — все остальные комбинации равны единице.

Теперь давайте перейдем к функции AND — создайте новый файл с именем `perceptron_and.py` и вставьте следующий код:

```

1 # импортируйте необходимые пакеты
2 из pyimagesearch.nn import Perceptron 3 import
numpy as np
4
5 # построить набор данных AND
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) 7 y = np.array([[0],
[0], [0], [1]])
8
9 # определяем наш персептрон и обучаем
его_

```

```

14 # теперь, когда нашперсепtron обучен, мы можем оценить его 15 print("[INFO] testing perceptron...")
16
17 # теперь, когда наша сеть обучена, выполните цикл по точкам данных 18 for
(x, target) в zip(X, y): # сделайте прогноз для точки данных и отобразите
18     результат # на нашей консоли
19
20
21     pred = p.predict(x)
22     print("[INFO] data={}, наземнаяправда={}, pred={}".format(x, target[0],
23         pred))

```

Обратите внимание, что единственное строки кода, которые изменились, это строки 6 и 7, где мы определяем набор данных И, а не набор данных ИЛИ.

Выполнив следующую команду, мы можем оценить персептрон по функции И:

```

$ python perceptron_and.py [INFO]
обучение персептрана...
[INFO] тестирование персептрана...
[ИНФОРМАЦИЯ] данные=[0 0], исх одна инфомация=0,
предыдущая=0 [ИНФОРМАЦИЯ] данные=[0 1], исх одна
инфомация=0, предыдущая=0 [ИНФОРМАЦИЯ] данные=[1 0],
правда=0, пред=0 [ИНФОРМАЦИЯ] данные=[1 1], основнаяправда=1, пред=1

```

Опять же, нашперсепtron смог правильно смоделировать функцию. Функция И верна только когда оба $x_0 = 1$ и $x_1 = 1$ – для всех остальных комбинаций побитового И равнотулю.

Наконец, давайте взглянем на нелинейно разделяемую функцию XOR внутри perceptron_xor.py:

```

1 # импортируем необходимые пакеты
2 из pyimagesearch.nn import Perceptron 3 import
numpy as np
4
5 # построить набор данных XOR
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) 7 y = np.array([[0], [1],
[1], [0]])
8
9 # определяем нашперсепtron и обучаем его_
10
11
12
13
14 # теперь, когда нашперсепtron обучен, мы можем оценить его 15
print("[INFO] testing perceptron...")
16
17 # теперь, когда наша сеть обучена, выполните цикл по точкам данных 18 for
(x, target) в zip(X, y): # сделайте прогноз для точки данных и отобразите
18     результат # на нашей консоли
19
20
21     pred = p.predict(x)
22     print("[INFO] data={}, наземнаяправда={}, pred={}".format(x, target[0],
23         pred))

```

Опять же, единственное строки кода, которые были изменены — это строки 6 и 7, где мы определяем XOR-данные. Оператор XOR истинен тогда и только тогда, когда один (но не оба) x равен единице.

Въполнив следующую команду, мы увидим, что персептрон не может изучить эту нелинейную связь:

```
$ python perceptron_xor.py [INFO]
обучение персептрана...
[INFO] тестирование персептрана...
[ИНФОРМАЦИЯ] данные=[0 0], исх однайинф ормация=0,
предыдущая=[1 [ИНФОРМАЦИЯ] данные=[0 1], основнаяинф ормация=1,
предыдущая=[1 [ИНФОРМАЦИЯ] данные=[1 0], исх однайинф ормация
правда=1, пред=0 [ИНФОРМАЦИЯ] данные=[1 1], основнаяправда=0, пред=0
```

Независимо от того, сколько раз вы запускаете этот эксперимент с различной скоростью обучения или разными схемами инициализации веса, в конечном итоге не сможете правильно смоделировать функцию XOR с помощью однослойного персептрана. Вместо этого нам нужно больше слоев, и с этим начинается глубокое обучение.

10.1.3 Обратное распространение и многоуровневые сети

Обратное распространение, возможно, является самым важным алгоритмом в истории нейронных сетей — без (эффективного) обратного распространения было бы невозможно обучить сети глубокого обучения на той глубине, которую мы наблюдаем сегодня. Обратное распространение можно считать краеугольным камнем современных нейронных сетей и глубокого обучения.

Первоначальное описание обратного распространения ошибки было введено еще в 1970-х годах, но только в основополагающей статье 1986 года «Обучение представлений с помощью ошибок обратного распространения» Румельхарта, Хигтона и Уильямса [16] мы смогли разработать более быстрый алгоритм лучше подходящий для обучения более глубоких сетей.

На сегодняшний день доступны буквально сотни (если не тысячи) руководств по обратному распространению ошибки. Вот некоторые из моих любимых: 1. Обсуждение Эндрю Нгу об обратном распространении в курсе машинного обучения от Coursera.

[76].

2. Сильно математически мотивированная Глава 2 — Как работает алгоритм обратного распространения

работы Майкла Нильсена «Нейронные сети и глубокое обучение» [108].

3. Стэнфордское исследование cs231n и анализ обратного распространения [57].

4. Отличный конкретный пример Мэтта Мазура (с реальными обработанными числами), демонстрирующий как работает обратное распространение [109].

Как видите, в руководствах по обратному распространению недостатка нет — вместо этого, чтобы изрыгать и повторять то, что было сказано другими сотни раз, я собираюсь использовать другой подход и сделать то, что делает публикации PyImageSearch особенно: создать интуитивно понятный, простая реализация алгоритма обратного распространения с использованием языка Python.

Внутри этой реализации мы создадим настоящую нейронную сеть и обучим ее с помощью алгоритма обратного распространения. К тому времени, как вы законите этот раздел, вы поймете, как работает обратное распространение ошибки, и, возможно, что более важно, у вас будет более четкое представление о том, как этот алгоритм используется для обучения нейронных сетей с нуля.

Обратное

распространение Алгоритм обратного распространения состоит из

двух фаз: 1. Прямой проход, когда наши выходные данные передаются через сеть и выводятся прогнозы полученные (также известный как этап распространения).

x	0	x	1	y	0	0
0	1			1		
1	0	1	1	1	0	

x	0	x	1	2	0	0
0	1			1		
1	0	1	1	1	1	

Таблица 10.2: Слева: набор данных побитового исключающего ИЛИ (включая метки классов). Справа: матрица дизайна набора данных XOR со вставленным столбцом смещения (исключающие метки классов для краткости).

2. Обратный проход, при котором мы вычисляем градиент функции потерь на последнем уровне (т. е. слое прогнозов) сети и используем его градиент для рекурсивного применения цепного правила для обновления весов в нашей сети (также известного как весовой коэффициент). этап обновления).

Мы начнем с рассмотрения каждого из этих этапов на высоком уровне. Оттуда мы реализуем алгоритм обратного распространения с использованием Python. После того, как мы реализовали обратное распространение, мы хотим иметь возможность делать прогнозы используя нашу сеть — это просто фаза прямого прохода, только с небольшой корректировкой (с точки зрения кода), чтобы сделать прогнозы более эффективными.

Наконец, я покажу, как обучить пользовательскую нейронную сеть с использованием обратного распространения ошибки и Python для 1. Набора данных XOR.

2. Набор данных МИСТ

Проход вперед

Целью прямого прохода является распространение наших входных данных по сети путем применения ряда скалярных произведений и активаций до тех пор, пока мы не достигнем всех одного слоя сети (т. е. наших прогнозов).

Чтобы визуализировать этот процесс, давайте сначала рассмотрим набор данных XOR (таблица 10.2, слева).

Здесь мы видим, что каждая запись X в матрице дизайна (слева) является сдвигом — каждая точка данных представлена двумя числами. Например, первая точка данных представлена вектором признаков (0,0), вторая точка данных - (0,1) и т. д. Поэтому у нас есть вектор однозначения в правом столбце.

Нашими целевыми векторами значениями являются метки классов. Учитывая все данные из матрицы проектирования наша цель состоит в том, чтобы правильно предсказать целевое векторное значение.

Как мы узнаем в разделе 10.1.3 ниже, для достижения идеальной точности классификации в этой задаче нам понадобится нейронная сеть с прямой связью, по крайней мере, с одним скрытым слоем, поэтому давайте начнем с 2 2 1 архитектура (рис. 10.9, вверху). Это означает, что мы можем использовать вектор признаков (0,0), (0,1) и (1,0) для каждого из трех классов. Мы можем либо:

1. Используйте отдельную переменную.
2. Рассматривайте смещение как обучаемый параметр в матрице весов, вставляя столбец единиц в вектор признаков.

Вставка столбца единиц в наш вектор признаков выполняется программно, чтобы убедиться, что мы понимаем этот момент, давайте обновим нашу матрицу проектирования XOR, чтобы явно увидеть, как это происходит (таблица 10.2, справа). Как видите, к нашим векторам признаков добавлен столбец единицами. На практике вы можете вставить этот столбец куда угодно, но мы будем размещать его либо как (1) первую запись в векторе признаков, либо (2) как последнюю запись в векторе признаков.

Поскольку мы изменили размер нашего вектора признаков (обычно включенного внутри самой реализации нейронной сети, так что нам не нужно явно изменять нашу матрицу проектирования), это меняет нашу (воспринимаемую) архитектуру сети с 2 2 1 на (внутренне) 3 3 1 (рис. 10.9, нижняя).

Мы по-прежнему будем называть эту сетевую архитектуру 2-2-1, но когда дело доходит до реализации, на самом деле это 3 3 1 из-за добавления члена смещения встроенного в матрицу весов.

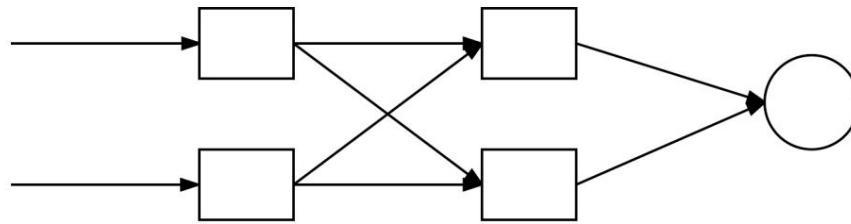
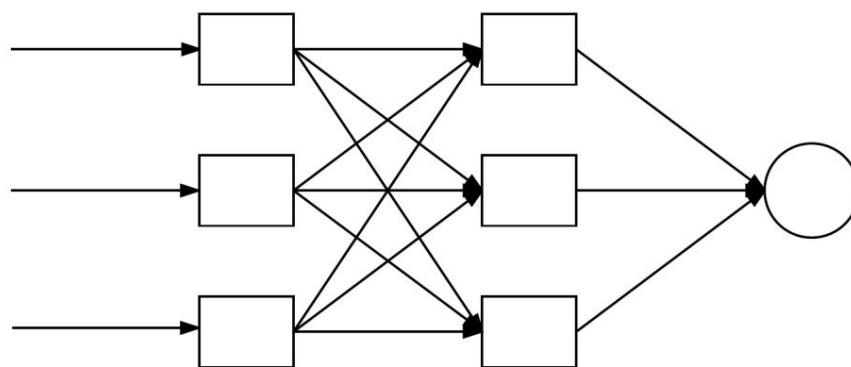
2-2-1**3-3-1**

Рисунок 10.9: Вверху: чтобы построить нейронную сеть для правильной классификации набора данных XOR, нам понадобится сеть с двумя входными узлами, двумя скрытыми узлами и одним выходным однозначным узлом. Это приводит к архитектуре 2-2-1. Внизу: наше фактическое представление архитектуры внутренней сети равно 3-3-1 из-за трюка смещения. В подавляющем большинстве реализаций нейронных сетей эта корректировка матрицы весов происходит внутри, и вам нечем беспокоиться, однако по-прежнему важно понимать, что происходит под капотом.

Наконец, чтобы для каждого слоя и для всех скрытых слоев требуется смещение; однако окончательный выход каждого слоя не требует смещения. Преимущество применения трюка смещения заключается в том, что нам больше не нужно явно отслеживать параметр смещения — теперь это обучаемый параметр в матрице весов, что делает обучение более эффективными значительно более простым в реализации. См. главу 9 для более подробного обсуждения этого, почему этот трюк с предвзятостью работает.

Чтобы увидеть прямой проход в действии, мы начали инициализировать веса в нашей сети, как показано на рис. 10.10. Обратите внимание, что каждая стрелка в матрице весов имеет связь с ней значение — это текущее значение веса для данного узла и означает величину, на которую данный ввод усиливается или уменьшается. Затем это значение веса будет обновлено на этапе обратного распространения.

В крайнем левом углу рисунка 10.10 мы представляем вектор признаков $(0, 1, 1)$ (и целевое значение 1 для сети). Здесь мы видим, что 0, 1 и 1 были назначены трем входным однозначным узлам в сети. Чтобы распространить значения по сети и получить окончательную классификацию, нам нужно взять скалярное произведение между всеми однозначными и значениями веса, а затем применить функцию активации (в данном случае сигмоидную функцию, σ).

Давайте вычислим все однозначные для двух узлов в скрытых слоях: 1.

$$\sigma((0 \times 0,351) + (1 \times 1,076) + (1 \times 1,116)) = 0,899$$

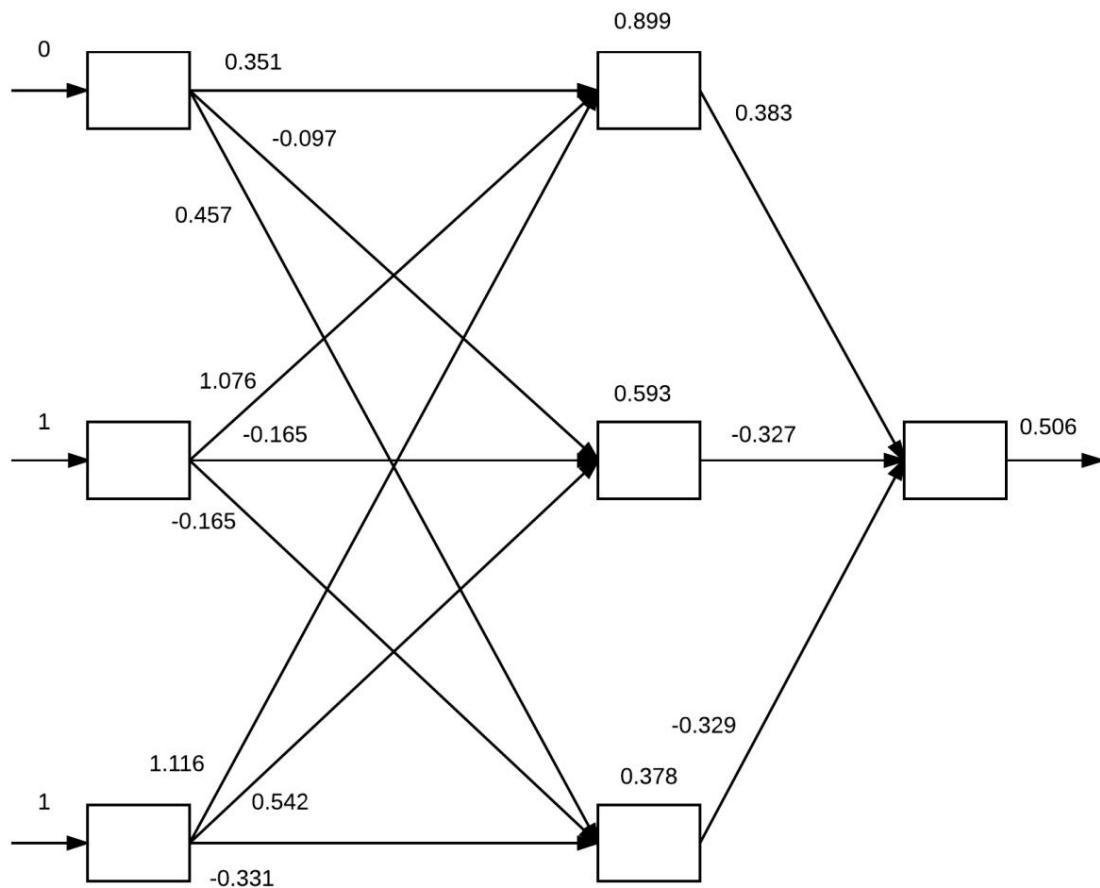


Рисунок 10.10: Пример прохода прямого распространения Вх одному вектору [0,1,1] передается в сеть. Берется скалярное произведение между вх одни данными и весами, после чего применяется сигмовидная функция активации для получения значений в скрытом слое (0,899, 0,593 и 0,378 соответственно). Наконец, скалярное произведение и сигмовидная функция активации вычисляются для последнего слоя, что дает результат 0,506. Применение ступенчатой функции к 0,506 дает 1, что действительно является правильной меткой целевого класса.

2. $\sigma((0 \times -0,097) + (1 \times -0,165) + (1 \times 0,542)) = 0,593$ 3. $\sigma((0 \times 0,457) + (1 \times -0,165) + (1 \times -0,331)) = 0,378$ Глядя на значения узлов скрытых слоев (рис. 10.10, средний), мы можем видеть, что узлы были обновлены, чтобы отразить наши вычисления.

Теперь у нас есть вх одни данные для узлов скрытого слоя. Чтобы вычислить вх одному прогнозу, мы должны снова вычислить скалярное произведение с последующей сигмовидной активацией:

$$\sigma((0,899 \times 0,383) + (0,593 \times -0,327) + (0,378 \times -0,329)) = 0,506 \quad (10.4)$$

Таким образом, вх од сети равен 0,506. Мы можем применить ступенчатую функцию, чтобы определить, является ли это вх од правильной классификацией или нет:

$$f(\text{net}) = \begin{cases} 0 & \text{иначе} \\ 1 & \text{если net} > 0 \end{cases}$$

Применив ступенчатую функцию с $\text{net} = 0,506$, мы видим, что наша сеть предсказывает 1, что на самом деле является правильной меткой класса. Однако наша сеть не очень уверена в этой метке класса — предсказанное значение 0,506 очень близко к порогу шага. В идеале этот прогноз должен быть ближе к 0,98–0,99, что означает, что наша сеть действительно изучила базовый шаблон в наборе данных. Чтобы наша сеть действительно «обучалась», нам нужно применить обратный проход.

Обратный проход

Чтобы применить алгоритм обратного распространения, наша функция активации должна быть дифференцируемой, чтобы мы могли вычислить частную производную ошибки по заданному вектору (E) , в k оду узла o_j и в k оду сети net_j .

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{i,j}} \quad (10.5)$$

Поскольку исчисление, стоящее за обратным распространением, было исчерпывающе объяснено много раз в предыдущих работах (см. Эндрю Нг [76], Майкл Нильсен [108], Мэтт Мэзур [109]), я собираюсь пропустить введение обновления правила цепочки обратного распространения в место этого объясните это с помощью кода в следующем разделе.

Для математически проницательных см. приведенные выше ссылки для получения дополнительной информации о цепном правиле и его роли в алгоритме обратного распространения. Объясняет этот процесс в коде, моя цель — помочь читателям понять обратное распространение через более интуитивный смысл реализации.

Реализация обратного распространения с помощью Python

Давайте приступим к реализации обратного распространения. Откройте новый файл, назовите его `neuronetwork.py` и приступайте к работе:

```

1 # импортируем необходимые пакеты
2
3 import numpy as np
4
5 class NeuralNetwork:
6     def __init__(self, layers, alpha=0.1):
7         # инициализируем
8         # список матриц весов, затем со временем # архитектуру сети и скорость
9         # обучения self.W = [] self.layers = слои self.alpha = alpha
10

```

В строке 2 мы импортируем единственный необходимый пакет, который нам понадобится для нашей реализации обратного распространения — библиотеку числовой обработки NumPy.

Затем в строке 5 определяется конструктор нашего класса `NeuralNetwork`. Конструктору требуется один аргумент, за которым следует второй обязательный аргумент:

- `Layers`: список целых чисел, который представляет реальную архитектуру сети прямого распространения. Например, значение `[2,2,1]` будет означать, что наш первый в x одной слой имеет два узла, наш скрытый слой имеет два узла, а наш конечный в k одной слой имеет один узел.

- `альфа`: здесь мы можем указать скорость обучения нашей нейронной сети. Это значение применяется на этапе обновления веса.

Строка 8 инициализирует наш список весов для каждого слоя `W`. Затем мы со временем слои и альфа в строках 9 и 10.

Наш список весов `W` пуст, поэтому давайте продолжим и инициализируем его сейчас:

```

12         # начать цикл с индекса первого слоя но
13         # остановимся не дойдя до последних двух слоев
14         для я в пр.arange(0, len(слои) - 2):
15             # случайным образом инициализируем весовую матрицу, соединяющую
16             # общее количество узлов в каждом соответствующем слое,
17             # добавляем дополнительный узел для смещения
18             w = np.random.randn(слои[i] + 1, слои[i + 1] + 1)
19             self.W.append(w/np.sqrt(слои[i]))

```

В строке 14 мы начинаем перебирать количество слоев в сети (т. е. `len(layers)`), но мы останавливаемся перед последними двумя слоями (мы узнаем, почему именно позже, в объяснении этого конструктора).

Каждый уровень в сети инициализируется случайным образом путем построения весовой матрицы $M \times N$ с помощью значений выборки из стандартного нормального распределения (строка 18). Матрица $M \times N$, так как мы хотим чтобы соединить каждый узел в текущем слое с каждым узлом в следующем слое.

Например, предположим, что `Layers[i] = 2` и `Layers[i + 1] = 2`. Наша весовая матрица поэтому будет 2×2 для соединения всех наборов узлов между слоями. Однако нам нужно однозначно быть осторожны так как мы забываем о важном компоненте — члене смещения. Для учета смещения мы добавляем единицу к количеству слоев `[i]` и слоев `[i + 1]` — это меняет наш вес матрица `w` и имеет форму 3×3 с учетом $2+1$ узлов для текущего слоя $2+1$ узлов для следующего слоя. Мы масштабируем `w` путем деления на квадратный корень из числа узлов в текущем слое, тем самым нормализуя дисперсию в выходе каждого нейрона [57] (строка 19).

Последний блок кода конструктора обрабатывает особые ситуации, когда для двух одних соединений требуется член смещения, но ввод не имеет:

```

21         # последние два слоя являются особым случаем, когда в них одни
22         # соединения между собой термине смещения, но ввод не имеется
23         w = np.random.randn(слои[-2] + 1, слои[-1])
24         self.W.append(w/np.sqrt(слои[-2]))

```

Отметим же, эти значения ввода выбираются случайным образом, а затем нормализуются. Следующая функция, которую мы определяем, — это «магический метод» Python с именем `_repr_` — эта функция полезна для отладки:

```

26     защищта __repr__(сам):
27         # создать и вернуть строку, представляющую сеть
28         # архитектура
29         вернуть «Нейронная сеть: {}». формат (
30             "-".join(str(l) для l в self.layers))

```

В нашем случае мы форматируем строку для нашего объекта `NeuralNetwork`, объединив целое число значение количества узлов в каждом слое. Учитывая значение слоев $(2, 2, 1)$, ввод введенный этой функции будет:

```

1 >>> из pyimagesearch.nn импорт NeuralNetwork
2 >>> nn = нейроннаясеть([2, 2, 1])
3 >>> напечатать(nn)
4 Нейронная сеть: 2-2-1

```

Далее мы можем определить нашу сигмовидную функцию активации:

```
32     определение сигмоида(я x):
33         # вычислить и вернуть значение активации сигмоида для
34         # заданное в x одно значение
35         вернуть 1,0 / (1 + np.exp(-x))
```

А также произведем одну сигмоиду которую мы будем использовать при обратном проходе:

```
37     определение sigmoid_deriv(я x):
38         # вычислить произведение сигмовидной функции ПРЕДПОЛАГАЕМ
39         # что 'x' уже прошел через 'сигмоид'
40         # функция
41         вернуть x * (1 - x)
```

Отметьте же, обратите внимание, что всякий раз, когда вы выполняете обратное распространение, всегда х отите выбрать функцию активации, которая является дифференцируемой.

Мы будем черпать вдохновение из библиотеки scikit-learn и определим функцию с именем `fit`, которая будет нести ответственность за фактическое обучение нашей нейронной сети:

```
43     def fit(self, X, y, epochs=1000, displayUpdate=100):
44         # вставить столбец из 1 в качестве последней записи в функции
45         # матрица -- эта маленькая интризость позволяет нам лечить смещение
46         # как обучаемый параметр в весовой матрице
47         X = np.c_[X, np.ones((X.shape[0]))]
48
49         # перебираем желаемое количество эпох
50         # для эпох и в пр.arange(0, эпохи):
51             # цикл по каждой отдельной точке данных и обучение
52             # наша сеть на нем
53             для(x, цель) в zip(X, y):
54                 self.fit_partial(x, цель)
55
56             # проверяем, должны ли мы отображать обновление обучения
57             если эпохи == 0 или (эпохи + 1) % displayUpdate == 0:
58                 потерь = self.calculate_loss(X, y)
59                 print("[INFO] epoch={}, loss={:.7f}".format(
60                     эпохи + 1, потерь))
```

Метод подгонки требует двух параметров, за которыми следуют два необязательных. Первый, `X`, наша `данные` тренировки. Второй, `y`, представляет собой соответствующие метки классов для каждой записи в `X`. Затем мы указываем `эпохи`, то есть количество эпох, для которых мы будем обучать нашу сеть. Параметр `displayUpdate` просто контролирует, сколько эпох мы будем печатать на нашем терминале.

В строке 47 мы выполняем трюк смещения вставляя столбец с единицами в качестве последней записи в нашей функции. матрица, `X`. Отсюда мы начинаем перебирать наше количество эпох в строке 50. Для каждой эпохи мы пройдемся по каждой отдельной точке данных в нашем тренировочном наборе, сделаем прогноз для точки данных, вычислить фазу обратного распространения, а затем обновить нашу матрицу весов (строки 53 и 54). Линии 57-60 просто проверьте, должны ли мы отображать обновление обучения на нашем терминале.

Фактическое сердце алгоритма обратного распространения находится внутри нашего метода `fit_partial`. Ниже:

```

62     def fit_partial(x, y):
63         # создаем наш список въек активаций для каждого слоя
64         # пока наша точка данных проходит через сеть; первое
65         # активация - это особый случай -- это просто ввод
66         # сам вектор признаков
67         A = [np.atleast_2d(x)]

```

Функция `fit_partial` требует два параметра:

- `x`: отдельная точка данных из нашей матрицы дизайна.
- `y`: соответствующая метка класса.

Затем мы инициализируем список `A` в строке 67 — этот список отвечает за хранение въек активаций для каждого уровня, поскольку наша точка данных `x` перед распространением по сети. Мы инициализируем этот список с `x`, который является простой точкой вх одных данных.

Отсюда мы можем начать фазу прямого распространения

```

69     # ОБРАТНАЯ СВЯЗЬ:
70     # ЦИКЛ ПО СЛОЯМ В СЕТИ
71     для слоя в range(0, len(self.W)):
72         # упраждаем активацию на текущем слое
73         # взвес скалярное произведение между активацией и
74         # весом яматрица -- это называется "чистый ввод"
75         # к текущему слою
76         сеть = A[слой].dot(self.W[слой])
77
78         # вычисление "чистого ввода" просто применяет наш
79         # нелинейная функция активации на чистый ввод
80         вх од = self.sigmoid(сеть)
81
82         # как только мы получим чистый ввод, добавляем его в наш список
83         # активаций
84         A. добавить(вх од)

```

Мы начинаем перебирать каждый слой в сети в строке 71. Сетевой вх од для текущего слоя винисляется путем скалярного произведения между матрицей активации и матрицей весов (строка 76). Чистый вх од текущего слоя затем винисляется путем пропускания чистого ввода через нелинейный сигмовидную активационную функцию. Получив чистый вх од, мы добавляем его в наш список активаций (строка 84).

Хотите верьте, хотите нет, но этот код представляет собой весь прямой проход, описанный в разделе 10.1.3 выше — мы просто перебираем каждый из слоев в сети, взвес скалярное произведение между активацией и весом, передавая значение через нелинейную функцию активации и продолжая на следующий слой. Таким образом, последняя запись в `A` является вх одом последнего слоя нашей сети (т. е. прогнозом).

Теперь, когда прямой проход выполнен, мы можем перейти к более сложному обратному проходу.

проход одят:

```

86     # ОБРАТНОЕ РАСПРОСТРАНЕНИЕ
87     # первая фаза обратного распространения заключается в вычислении
88     # разница между нашим *прогнозом* (конечный результат
89     # активации в списке активаций) и истинная цель
90     # ценность

```

```

91         ошибка = A[-1] - у
92
93             # от сюда нам нужно применить цепное правило и построить наш
94             # список делт 'D'; первая запись в делтах
95             # просто произведение ошибки в вк одного слоя на произв одну
96             # нашей функции активации для вк однозначения
97             D = [ошибка * self.sigmoid_deriv(A[-1])]
```

Первая фаза обратного прохода состоит в том, чтобы вычислить нашу ошибку или просторазницу между наша предсказанная метка и метка достоверности (строка 91). С момента последней записи в списке активаций

А содержит ввод сети, мы можем получить доступ к вк одному прогнозу через $A[-1]$. Значение y – целевой ввод для точек вх однък данък x .

 При использовании языка программирования Python указание значения индекса -1 указывает на то, что мы получим доступ к последней записи в списке. Вы можете прочитать больше о массиве Python индексами срезов в этом руководстве: <http://pyimg.co/6dfae>.

Затем нам нужно начать применять цепное правило, чтобы построить наш список делта, D . Делты будут использоваться для обновления наших весовых матриц, масштабированием скорости обучения алгоритма. Первый вх од в делты $list$ – это ошибка нашего вк одного слоя умноженная на произв одну сигмоиды для вк однозначения (строка 97).

Учитывая делту для последнего слоя сети, теперь мы можем работать в обратном направлении, используя цикл `for`:

```

99     # как только вы поймете цепное правило, все станет очень просто
100    # для реализации с помощью цикла for -- просто переберите
101    # слои в обратном порядке (игнорируя последние два, так как мы
102    # уже учли их)
103    для слоя в np.arange(len(A) - 2, 0, -1):
104        # делта для текущего слоя равна делте
105        № *предыдущего слоя*, отмеченного матрицей весов
106        # текущего слоя с последнюю ширину умножением делты
107        # по произв одной нелинейной функции активации
108        # для активаций текущего слоя
109        делта = D[-1].dot(self.W[слой].T)
110        делта = делта * self.sigmoid_deriv(A [слой])
111        D.append(делта)
```

В строке 103 мы начинаем перебирать каждый слой в сети (игнорируя предыдущий два слоя как они уже учтены в строке 97) в обратном порядке как нам нужно работать назад, чтобы вычислить делта-обновления для каждого слоя. Делта для текущего слоя равен делта предыдущего слоя $D[-1]$ с точками весовой матрицы текущего слоя (строка 109).

Чтобы закончить вычисление делты, мы умножаем ее, передавая активацию для слоя через нашу произв одну от сигмоиды (строка 110). Затем мы обновляем список делта D с помощью делты мы только что вычислили (строка 111).

Глядя на этот блок кода, мы видим, что шаг обратного распространения является итеративным — мы просто беря делту из предыдущего слоя, расставляя точки с весами текущего слоя, и затем умножение на производную от активации. Этот процесс повторяется до тех пор, пока мы не достигнем первого слоя в сети.

Учитывая наш список делта D , мы можем перейти к этапу обновления веса:

```

113         # так как мызациклились на наших слоях в обратном порядке, нам нужно
114         # поменять местами дельты
115     D = D[::-1]
116
117     # ЭТАП ОБНОВЛЕНИЯ ВЕСОВ
118     # ЦИКЛ ПО СЛОЯМ
119     для слоя в пр.arange(0, len(self.W)):
120         # обновить наши веса, взяв скалярное произведение слоя
121         # активации с соответствующими дельтами, затем умножение
122         # это значение с небольшой скоростью обучения добавлением к нашему
123         # весовой матрица -- здесь происходит фактическое "обучение"
124         # место
125         self.W[слой] += -self.alpha * A[слой].T.dot(D[слой])

```

Имейте в виду, что на этапе обратного распространения мызациклились на наших слоях в обратном порядке. Чтобы выполнить нашу фазу обновления веса, мы просто изменим порядок записей в D, чтобы мы могли цикл по каждому слою последовательно от 0 до N, общее количество слоев в сети (строка 115).

Обновление нашей фактической весовой матрицы (т. е. того, где происходит фактическое «обучение») выполнено. На линии 125, которая является нашим градиентом спуском. Мы берем скалярное произведение активации текущего слоя, A[layer] с дельтами текущего слоя D[layer] и умножить их на скорость обучения алфава. Это значение добавляется к матрице весов для текущего слоя W[layer].

Мы повторяя этот процесс для всех слоев в сети. После выполнения фазы обновления веса обратное распространение официально сделано.

Как только наша сеть будет обучена на заданном наборе данных, мы можем сделать прогнозы по тестированию. set, что может быть выполнено с помощью метода прогнозирования ниже:

```

127     Def Predict (self, X, addBias=True):
128         # инициализируем вектор одной прогноз как в х одни объекты-- это
129         # значение будет (вперед) распространяться по сети на
130         # получить окончательный прогноз
131         p = пр.покрайней_мере_2d (X)
132
133         # проверяем, нужно ли добавлять столбец смещения
134         если добавить смещение:
135             # вставить столбец из 1 в качестве последней записи в функции
136             # матрица (смещение)
137             p = np.c_[p, np.ones((p.shape[0]))]
138
139         # зацикливаемся на наших слоях в сети
140         для слоя в пр.arange(0, len(self.W)):
141             # выполнить вектор одной прогноз так же просто, как взять
142             # скалярное произведение между текущим значением активации 'p'
143             # и весовой матрица, связанный с текущим слоем,
144             # затем пропускаем это значение через нелинейную активацию
145             # функция
146             p = self.sigmoid (np.dot (p, self.W [слой]))
147
148         # вернуть предсказанное значение
149         вернуть p

```

Функция предсказания — это просто про славленный прямой проход. Эта функция принимает один требуемый параметр, за которым следует второй необязательный:

- X: точки данных, для которых мы будем прогнозировать метки классов.

- `addBias`: логическое значение, указывающее, нужно ли нам добавить столбец единиц к X , чтобы въполнить предвзятый трюк.

В строке 131 мы инициализируем r , вък однъе прогнозы как точки вък однък данък X . Это значение r будет передаваться через каждый слой в сети, распространяться, пока мы не достигнем окончательного вък одного прогноза.

В строках 134-137 мы проверяем, следует ли вводить член смещения в точки данък. Если это так, мыставляем столбец с единицами в качестве последнего столбца в матрице (точнотак же, как мыделали в методе подгонки въще).

Оттуда мывъполняем прямое распространение, перебирая все слои в нашей сети в строке 140. Точки данък r обновляются путем възяя скалярного произведения между текущими активациями r и матрицей весов для текущего слоя с последующей передачей вък однък данък r через нашу сигмовидную функцию активации (строка 146).

Учитывая, что мызацикливаемся въх слоях в сети, мыв конечном итоге достигнем последнего слоя, который даст нам нашокончательный прогноз метки класса. Мыозвращаем предсказанное значение въввывающу функии в строке 149.

Последняя функция, которую мыопределим в классе `NeuralNetwork`, будет использоваться для вычисления потери във семнадцатом тренировочном наборе:

```

151     def calculate_loss(self, X, target): # делаем прогнозы для точек
152         въх однък данък, затем вънисляем # цели потерь = np.atleast_2d(targets).Predicts =
153         self.predict(X, addBias=False) loss = 0,5 * np.sum ((прогнозы-цели) ** 2)
154
155
156
157
158         # вернуть проигрыш
159         обратная потеря

```

Функция `calculate_loss` требует, чтобы мыпередавали точки данък X вместе с их метками истинности, целями. Мыделаем прогнозы для X в строке 155, а затем вънисляем вадрат суммы ошибок в строке 156. Затем потери возвращаются въввывающу функии в строке 159. Помимо обучения нашей сети мыдолжны идеть, что эти потери уменьшаются

Обратное распространение с помощью Python Пример № 1: побитовое XOR

Теперь, когда мыреализовали нашкласс `NeuralNetwork`, давайтепродолжим и обучим его на наборе данък побитового XOR. Как мызнаем из нашей работы с `Perceptron`, этот набор данък не являетсялинейно разделимым — наша цель будет состоять в том, чтобы обучить нейронную сеть, которая может моделировать эту нелинейную функцию.

Откройте новый файл, назовите его `pp_xor.py` и вставьте следующий код:

```

1 # импортируйте необходимые пакеты2 из
2 pyimagesearch.nn import NeuralNetwork3 import numpy as np
3
4
5 # построить набор данък XOR 6 X = np.array([[0,
6 [0, 1], [1, 0], [1, 1]]) 7 y = np.array([[0], [1], [1], [0]])

```

Строки 2 и 3 импортируют необходимые пакеты Python. Обратите внимание, как мыимпортируем нашедавно реализованный класс `NeuralNetwork`. Затем в строках 6 и 7 создается набор данък XOR, как показано в таблице 10.1 ранее в этой главе.

Теперь мыможем определить нашу сетевую архитектуру и обучить ее:

```
9 # определяем нашу нейронную сеть 2-2-1 и обучаем ее
10 nn = нейронная сеть ([2, 2, 1], альфа = 0,5) 11 nn.fit
(X, y, эпох и = 20000)
```

В строке 10 мы создаем экземпляр нашей нейронной сети с архитектурой 2-2-1, что подразумевает наличие

1. Вход один слой с двумя узлами (т.е. наши два входа).
2. Один скрытый слой с двумя узлами.
3. Выход одной слой с одним узлом.

Линия 11 обучает нашу сеть в общей сложности 20 000 эпох.

Как только наша сеть будет обучена, мы перейдем к нашим наборам данных XOR, позволив сети предсказать выход для каждого из них и отображать прогноз на нашем экране:

```
13 # теперь, когда наша сеть обучена, выполните цикл по точкам данных XOR 14
for (x, target) в zip(X, y): # сделайте прогноз для точки данных и отобразите
    результат # на нашей консоли
15
16
17     pred = nn.predict(x)[0][0] step =
18     1 if pred > 0.5 else 0 print("[INFO]")
19     data={}, Ground-Truth={}, pred={:.4f}, step ={}".format( x, target[0], pred, step))
20
```

Строка 18 применяет ступенчатую функцию к сигмовидному выходу. Если прогноз > 0,5, мы вернем единицу, в противном случае вернем ноль. Применение этой шаговой функции позволяет нам бинаризовать наши метки выхода одного класса, как и функция XOR.

Чтобы обучить нашу нейронную сеть с использованием обратного распространения с помощью Python, просто выполните следующие команды:

```
$ python nn_xor.py
[ИНФОРМАЦИЯ] эпох а = 1, потеря =
0,5092796 [ИНФОРМАЦИЯ] эпох а = 100, потеря =
= 0,4923591 [ИНФОРМАЦИЯ] эпох а = 200, потеря = 0,46777865
...
[INFO] эпох а = 19800, потеря = 0,0002478
[INFO] эпох а = 19900, потеря = 0,0002465
[INFO] эпох а = 20000, потеря = 0,0002452
```

График квадрата потерь показан ниже (рис. 10.11). Как мы видим, в процессе обучения потери медленно уменьшаются примерно до нуля. Кроме того, глядя на последние четыре строки выхода, мы можем увидеть наши прогнозы.

```
[ИНФОРМАЦИЯ] данные=[0 0], основная правда=0, пред=0,0054, шаг=0
[ИНФОРМАЦИЯ] данные=[0 1], основная правда=1, пред=0,9894, шаг=1
[ИНФОРМАЦИЯ] данные=[1 0], основная правда = 1, пред = 0,9876, шаг = 1
[ИНФОРМАЦИЯ] данные=[1 1], основная правда = 0, пред = 0,0140, шаг = 0
```

Для каждой точки данных наша нейронная сеть смогла правильно изучить шаблон XOR, демонстрируя что наша многослойная нейронная сеть способна изучать нелинейные функции.

Чтобы продемонстрировать, что для изучения функции XOR требуется как минимум один скрытый слой, вернитесь к строке 10, где мы определяем архитектуру 2-2-1:



Рисунок 10.11: Потери вовремени длянашейнейроннойсети 2-2-1.

10 # определяем нашу нейронную сеть 2-2-1 и обучаем ее

11 nn = нейроннаясеть ([2, 2, 1], альфа = 0,5) 12
nn.fit (X, y, эпох и = 20000)

И измените егонаархитектуру 2-1:

10 определить нашу нейронную сеть 2-1 и обучить ее

11 nn = нейроннаясеть ([2, 1], альфа = 0,5) 12
nn.fit (X, y, эпох и = 20000)

Оттуда выможетепопыгатьсяпереобучитьсвоюсеть:

\$ питон nn_xor.py

...

[ИНФ О] данные=[0 0], основнаяправда=0, пред=0,5161, шаг=1
[ИНФ О] данные=[0 1], основнаяправда=1, пред=0,5000, шаг=1
[ИНФ О] данные = [1 0], основнаяправда = 1, пред = 0,4839, шаг = 0
[ИНФ О] данные = [1 1], основнаяправда = 0, пред = 0,4678, шаг = 0

Независимо от того, сколько вызовитесь соскоростью обученияили инициализацией веса, выникогда не сможете аппроксимировать функцию XOR. ИменноНПОТУМО многослойныесети с нелинейнымифункциямиактивации, обучениями с помощью обратногораспространения так важны— они позволяют намизучать закономерности в наборах данных, которые в противномслучае были бынелинейноразделимы

Обратное распространение с помощью Python Пример: пример MNIST В качестве второго, более интересного примера, давайте рассмотрим подмножество набора данных MNIST (рис. 10.12) для распознавания рукописных цифр. Это подмножество набора данных MNISTстроено в библиотеку scikit-learn и включает 1797 примеров цифр, каждая из которых представляет собой изображения градаций серого 8×8 (их одни изображения имеют размер 28×28). При сглаживании эти изображения представляются $8 \times 8 = 64$ -мерный вектор.

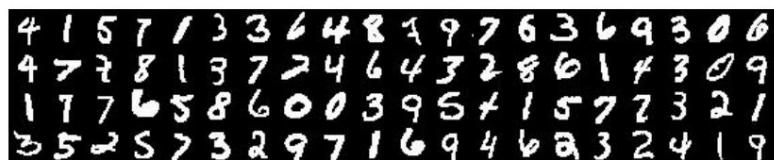


Рисунок 10.12: Пример набора данных MNIST. Цель этого набора данных — правильную классифицировать рукописные цифры от 0 до 9.

Давайте продолжим обучение реализацию NeuralNetwork на этом подмножестве MNIST. Открытым создайте новый файл, назовите его nn_mnist.py, и мы приступим к работе:

```
1 # импортируем необходимые пакеты
из pyimagesearch.nn импортируем NeuralNetwork
из sklearn.preprocessing импортируем LabelBinarizer
из sklearn.model_selection импортируем train_test_split
импортируем classification_report
из sklearn импортируем наборы
данных
```

Мы начинаем с строк 2-6, импортируя необходимые пакеты Python.

Оттуда мы загружаем набор данных MNIST с диска, используя спомогательные функции scikit-learn:

```
8 # загрузите набор данных MNIST и примените минимальное/максимальное
# масштабирование, чтобы масштабировать значение интенсивности пикселей
# в диапазоне [0, 1] (каждое изображение 10 # представлено вектором признаков
# 8 x 8 = 64)
9 print("[INFO] загружает набор данных MNIST (образец...)")
```

10 цифры =

```
datasets.load_digits()
11 data = digits.data.astype("float")
12 data = (data - data.min()) /
    (data.max() - data.min())
13 print("[INFO] образцы {}, dim: {}".format(data.shape[0],
    data.shape[1]))
```

16

Мы также выполняем минимальную/максимальную нормализацию, масштабируя каждую цифру в диапазоне [0,1] (строка 14).

Далее, давайте построим разделение обучения/тестирования, используя 75% данных для тестирования и 25% для оценки:

```
18 # построить тренировочные и тестовые сплиты
19 (trainX, testX, trainY, testY) = train_test_split(data,
20     цифрыцель, test_size=0,25)
21
22 # преобразовать метки из целых чисел в векторы
23 trainY = LabelBinarizer().fit_transform(trainY)
24 testY =
LabelBinarizer().fit_transform(testY)
```

Мы также будем кодировать целые числа методом нашего класса в виде векторов. Этот процесс называется **горячим кодированием**, который мы мы подробно обсудим позже в этой главе.

Оттуда мы хотим обучить нашу сеть:

```
26 # обучаем сеть
27 print("[INFO] обучаясь...")
28 nn = NeuralNetwork([trainX.shape[1], 32, 16, 10])
29 print("[INFO] {}".format(nn))
30 nn.fit(trainX, trainY, эпох=1000)
```

Здесь мы видим, что обучаем нейронную сеть с архитектурой 64-32-16-10.

В $\text{в} \times \text{в}$ одной слой имеет десять узлов из-за того, что существует десять возможных в $\text{в} \times \text{в}$ одних классов для цифр 0-9.

Затем мы позвольяем нашей сети обучаться в течение 1000 эпох. Как только наша сеть будет обучена, мы можем оценить на тестовом наборе:

```
32 # оценка сети 33 print("[INFO]  
оценка сети...") 34 предсказания=nn.predict(testX) 35  
предсказания=предсказания.argmax(axis=1) 36  
print(classification_report(testY.argmax(axis=1), предсказания))
```

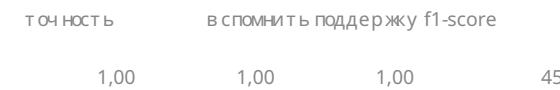
Строка 34 в `nnislyat` в `yk` однъе прогнозы для каждой точки данных в `testX`. Массив прогнозов имеет форму $(450, 10)$, поскольку в тестовом наборе имеется 450 точек данных, каждая из которых имеет десять возможных вероятностей меток класса.

Чтобы найти метку класса с наибольшей вероятностью для каждой точки данных, мы используем функцию `argmax` в строке 35 — эта функция вернет индекс метки с наибольшей прогнозируемой вероятностью. Затем мы отображаем хоруло отформатированный отчет о классификации на нашем экране в строке 36.

Чтобы обучить нашу пользовательскую реализацию NeuralNetwork на наборе данных MNIST, просто выполните следующую команду:

```
$ python nn_mnist.py [INFO]
загрузка набора данных MNIST (пример)...
[И НФ] проб: 1797, размер: 64 [И НФ]
обучаю шасть...
[И НФ] NeuralNetwork: 64-32-16-10 [И НФ]
эпох а = 1, пот еръ= 604,5868589 [И НФ] эпох а
= 100, пот еръ= 9,1163376 [И НФ] эпох а = 200,
пот еръ= 3,7157723 [И НФ] эпох а = 300, пот еръ
= 3,7157723 =2,6078803 [INFO] эпох а=400,
пот еръ=2,3823153 [INFO] эпох а=500,
пот еръ=1,8420944 [INFO] эпох а=600,
пот еръ=1,3214138 [INFO] эпох а=700,
пот еръ=1,2095033 [INFO] эпох а=800, пот еръ
=1,1663942 [INFO] эпох а=900, loss=1,1394731
[INFO] эпох а=1000, loss=1,1203779 [INFO]
оценка сет...

```



1	0,98	1,00	0,99	51
2	0,98	1,00	0,99	47
3	0,98	0,93	0,95	43
4	0,95	1,00	0,97	39
5	0,94	0,97	0,96	35
6	1,00	1,00	1,00	53
7	1,00	1,00	1,00	49
8	0,97	0,95	0,96	41
9	1,00	0,96	0,98	47
среднее / общее	0,98	0,98	0,98	450

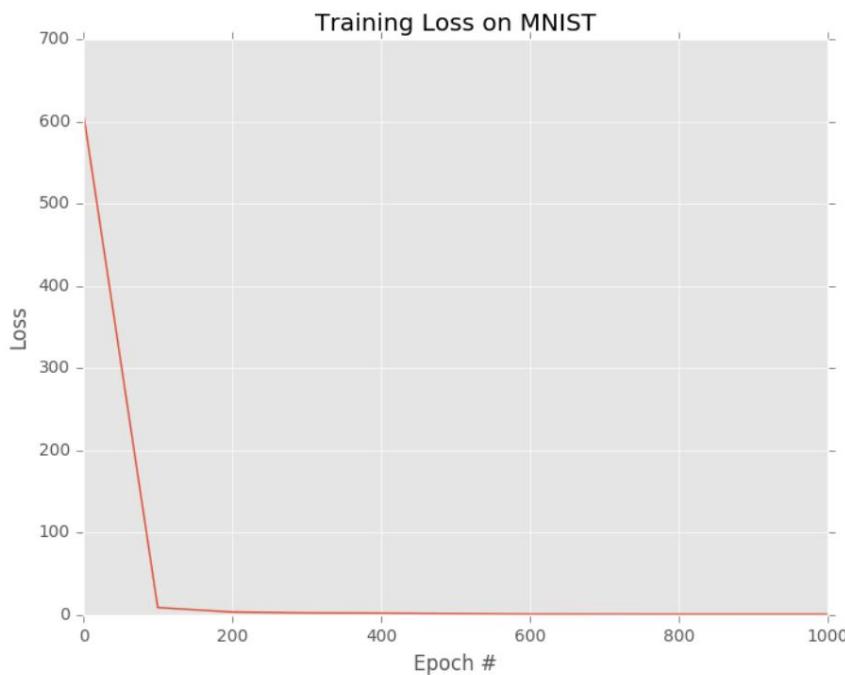


Рисунок 10.13: График потерь при обучении в наборе данных MNIST с использованием прямой связи 64-32-16-10 нейронная сеть.

Я также включил график квадрата потерь (рис. 10.13). Обратите внимание, как начинается наша потеря очень высока, но быстро падает в процессе обучения. Наша точность классификации демонстрирует, что мы получаем точность классификации 98% на нашем тестовом наборе; тем не менее, у нас есть некоторые проблемы с классификацией цифр 4 и 5 (точность 95% и 94% соответственно). Далее в этой книге мы узнаем, как обучать сверточные нейронные сети на полном наборе данных MNIST и улучшать наши точности дальше.

Сводка по обратному распространению

В этом разделе мы узнали, как реализовать алгоритм обратного распространения с нуля, используя Python. Обратное распространение — это обобщение семейства алгоритмов градиентного спуска, которое специально используется для обучения многоуровневых сетей прямого распространения.

Алгоритм обратного распространения состоит из двух фаз:

1. Прямой проход, когда мы передаем наши вх однъе данъе через сеть, чтобы получить наши вх однъе данъе классификации.

2. Обратный проход (т. е. этап обновления веса), когда мы вычисляем градиент функции потерь и используем эту информацию для итеративного применения цепного правила для обновления весов в нашей сети.

Независимо от того, работаем мы с простыми нейронными сетями с прямой связью или сложными, глубокими с вложенными нейронными сетями, алгоритм обратного распространения по-прежнему используется для обучения этих моделей. Это достигается путем обеспечения дифференцируемости функций активации внутри сети, что позволяет применять цепное правило. Кроме того, любые другие слои внутри сети, требующие обновления своих весов/параметров, также должны быть совместимы с обратным распространением.

Мы реализовали наш алгоритм обратного распространения с использованием языка программирования Python и разработали многоуровневый класс NeuralNetwork с прямой связью. Затем эта реализация была обучена на наборе данных XOR, чтобы продемонстрировать, что наша нейронная сеть способна изучать нелинейные функции, применяя алгоритм обратного распространения по крайней мере с одним скрытым слоем. Затем мы применили ту же реализацию обратного распространения+Python к подмножеству набора данных MNIST, чтобы продемонстрировать, что алгоритм можно использовать и для работы с данными изображений.

На практике обратное распространение может быть не только сложным в реализации (из-за ошибок в вычислении градиента), но также трудно сделать его эффективным без специальных библиотек оптимизации, поэтому мы часто используем такие библиотеки, как Keras, TensorFlow и mxnet, которые имеют уже (правильно) реализовано обратное распространение с использованием оптимизированных стратегий.

10.1.4 Многоуровневые сети с Keras

Теперь, когда мы реализовали нейронные сети на чистом Python, давайте перейдем к предпочтительному методу реализации — использованию специальной (высокооптимизированной) библиотеки нейронных сетей, такой как Keras.

В следующих двух разделах я расскажу, как реализовать многослойные сети с прямой связью и применить их к наборам данных MNIST и CIFAR-10. Этот результат вряд ли будет «современным», но послужит двум целям:

- Продемонстрировать, как вы можете реализовать простые нейронные сети с помощью библиотеки Keras.
- Получите базовый уровень, используя стандартные нейронные сети, которые мы можем сравнить со сверточной нейронной сетью (отметим, что CNN значительно превзойдет наши предыдущие методы).

МИСТ

В разделе 10.1.3 выше мы использовали только образец набора данных MNIST под двумя причинами:

- Продемонстрировать, как реализовать вашу первую нейронную сеть с прямой связью на чистом Python.
- Для ускорения сбора результатов — учитывая что наша реализация на чистом Python определенно не оптимизирована, это займет больше времени.

Поэтому мы использовали образец набора данных. В этом разделе мы будем использовать полный набор данных MNIST, состоящий из 70 000 точек данных (7 000 примеров на цифру). Каждая точка данных представлена 784-дневным вектором, соответствующим (сглаженным) изображениям 28×28 в наборе данных MNIST. Наша цель — обучить нейронную сеть (используя Keras) для получения > 90% точности в этом наборе данных.

Как мы видим, использовать Keras для построения нашей сетевой архитектуры значительно проще, чем нашу чистую версию Python. На самом деле реальная сетевая архитектура будет занимать всего четверть строки кода — остальная часть кода в этом примере просто включает загрузку данных с диска, преобразование меток классов и последующее отображение результатов.

Для начала откройте новый файл, назовите его keras_mnist.py и вставьте следующий код:

¹ # импортируем необходимые пакеты
из sklearn.preprocessing import LabelBinarizer

```
3 из sklearn.model_selection импортируем train_test_split 4 из
sklearn.metrics импортируем Classification_report 5 из keras.models
импортируем Sequential 6 из keras.layers.core импортируем
Dense 7 из keras.optimizers импортируем SGD 8 из sklearn
импортируем наборы данных 9 импортируем matplotlib.pyplot
как plt 10 импортируем numpy как np 11 импортируем argparse
```

Строки 2-11 импортируют необязательные пакеты Python. LabelBinarizer будет использоваться для горячего кодирования наших целочисленных меток в виде векторных меток. Горячее кодирование преобразует категориальные метки из одного целого числа в вектор. Многие алгоритмы машинного обучения (включая нейронные сети) выигрывают от этого типа представления меток. Я буду обсуждать однократное кодирование более подробно и приводить несколько примеров (включая использование LabelBinarizer) позже в этом разделе.

Train_test_split в строке 3 будет использоваться для создания наших тренировочных и тестовых разделов из набора данных MNIST. Функция classification_report даст нам хорошо оформленный отчет, отображающий общую точность нашей модели, а также разбивку по точности классификации для каждой цифры.

Строки 5-7 импортируют необязательные пакеты для создания простой нейронной сети с прямой связью с помощью Keras. Класс Sequential указывает, что наша сеть будет иметь прямую связь, и слои будут добавляться в класс последовательно, один поверх другого. Класс Dense в строке 6 — это реализация наших полно связанных слоев. Чтобы наша сеть действительно обучалась, нам нужно применить SGD (строка 7) для оптимизации параметров сети. Наконец, чтобы получить доступ к полному набору данных MNIST, нам нужно импортировать помощник наборов данных из scikit-learn в строке 8.

Давайте перейдем к разбору наших аргументов командной строки:

```
13 # построить разбор аргумента и разобрать аргументы 14 ap =
argparse.ArgumentParser() 15 ap.add_argument("-o", "--output",
required=True,
16     help="путь к выходному графику потерю/точности") 17
args = vars(ap.parse_args())
```

Здесь нам нужно только один переключатель, --output, который является путем, по которому наша цифра, отображающая потерю и точность с течением времени, будет сохранена на диске.

Далее давайте загрузим полный набор данных MNIST:

```
19 # получить набор данных MNIST (если вы запускаете этот скрипт 20 # впервые,
загрузка может занять минуту -- будет загружен набор данных MNIST размером 55 МБ
21 #) 22 print("[INFO] loading MNIST (полный) набор данных ...") 23 набор данных =
datasets.fetch_mldata("Оригинал MNIST")
```

```
24
25 # масштабируем небольшие изображения и генерируем пиксели до диапазона [0,
1.0], затем 26 # строим разбиения для обучения и тестирования 27 data =
dataset.data.astype("float") / 255.0 28 (trainX, testX, trainY, testY) = train_test_split
(данные,
29     набор данных .цель, test_size=0.25)
```

Строка 23 загружает набор данных MNIST с диска. Если вы никогда раньше не запускали эту функцию, набор данных MNIST будет загружен со временем локально на вашем компьютере — эта загрузка составляет 55 МБ.

и может занять минуту или две, чтобы завершить загрузку, в зависимости от вашего интернет-соединения. После загрузки набора данных он кэшируется на вашем компьютере, и его не нужно будет загружать снова.

Затем мы выполним нормализацию данных в строке 27, масштабируя интенсивность пикселей до диапазона [0,1]. Мы создаем разделение обучения и тестирования используя 75% данных для обучения и 25% для тестирования в строках 28 и 29.

Учитывая разделение обучения и тестирования теперь мы можем кодировать наши метки:

```
31 # преобразовать метки из целых чисел в векторы32
lb = LabelBinarizer() 33 trainY = lb.fit_transform(trainY) 34 testY =
lb.transform(testY)
```

Каждая точка данных в наборе данных MNIST имеет целочисленную метку в диапазоне [0,9], по одной для каждой из десяти возможных цифр в наборе данных MNIST. Метка с значением 0 указывает, что соответствующее изображение содержит нулевую цифру. Точно так же метка с значением 8 указывает, что соответствующее изображение содержит число восемь.

Однако сначала нам нужно преобразовать эти целочисленные метки в векторные метки, где индекс в вектор для метки устанавливается равным 1 и 0 в противном случае (этот процесс называется горячим кодированием).

Например, рассмотрим метку 3, и мы хотим выполнить ее бинаризацию/горячее кодирование — метка 3 теперь становится

[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

Обратите внимание, что только индекс для цифры три установлен равным единице, а все остальные элементы вектора установлены равными нулю. Проницательные читатели могут удивиться почему обновляется четвертая а не третья запись в векторе? Напомним, что первая запись в метке фактически соответствует цифре нуль. Следовательно, запись для цифры три фактически является четвертым индексом в списке.

Вот второй пример, на этот раз с бинарной меткой 1:

[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

Второй элемент вектора устанавливается равным единице (поскольку первый элемент соответствует метке 0), а все остальные элементы устанавливаются равными нулю.

Являю чил однократное кодирование для каждой цифры 0-9, в листинг ниже:

```
0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] 1: [0, 1, 0, 0,
0, 0, 0, 0, 0] 2: [0 , 0, 1, 0, 0, 0, 0, 0, 0,
0] 3: [0, 0, 0, 1, 0, 0, 0, 0, 0] 4: [0, 0,
0 , 0, 1, 0, 0, 0, 0] 5: [0, 0, 0, 0, 0, 1, 0,
0, 0, 0] 6: [0, 0, 0, 0, 0 , 1, 0, 0, 0] 7:
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0] 8: [0, 0, 0, 0,
0, 0 , 0, 1, 0, 0] 9: [0, 0, 0, 0, 0, 0, 0, 0, 1]
```

Это кодирование может показаться утомительным, но многие алгоритмы машинного обучения (включая нейронные сети) варьируют от такого представления меток. К счастью, большинство пакетов программного обеспечения для машинного обучения предоставляют метод/функцию для выполнения «горячего» кодирования избавляя от большей части утомительной работы.

Строки 32-34 просто выполняют этот процесс прямого кодирования всех одноклассенных меток в виде вектора.

метки как для обучения, так и для тестового набора.

Далее давайте определим нашу сетевую архитектуру:

```
36 # определяем архитектуру 784-256-128-10 с помощью Keras
37 _ _ 128, активация="sigmod"))
38 40 model.add(Dense(10,
39   активация="softmax"))
```

Как видите, наша сеть представляет собой архитектуру с прямой связью, созданную классом Sequential в строке 37. Эта архитектура подразумевает, что слои будут располагаться один над другим, а вък одни данные предыдущего слоя передаются следующему.

Строка 38 определяет первый полно связанный уровень в сети. Для input_shape установлено значение 784, размерность каждой точки данных MNIST. Затем мы изучаем 256 весов в этом слое и применяем сигмовидную функцию активации. Следующий слой (строка 39) изучает 128 весов. Наконец, в строке 40 применяется один полно связанный слой, на этот раз изучающий только 10 весов, соответствующих десяти (0-9) вък одному классу. Вместо сигмовидной активации мы будем использовать активацию softmax для получения нормализованных вероятностей классов для каждого прогноза.

Давайте продолжим обучение нашей сети:

```
42 # обучить модель с помощью
43 SGD 43 print("[INFO] training network...") 44
44 sgd = SGD(0.01) 45
45 model.compile(loss="categorical_crossentropy", optimizer=sgd,
46   метрики=["точность"])
47 H = model.fit(trainX, trainY, validation_data=(testX, testY),
48   эпохи = 100, размер партии = 128)
```

В строке 44 мы инициализируем оптимизатор SGD со скоростью обучения 0,01 (которую обычно можно записать как 1e-2). Мы будем использовать функцию кросс-энтропии для потерь категорий в качестве нашей метрики потерь (строки 45 и 46). Использование функции потери кросс-энтропии также является причиной того, что нам пришлось преобразовать наши целочисленные метки в векторные метки.

В ввод .fit модели в строках 47 и 48 запускает обучение нашей нейронной сети.

Мы предоставим обучающие данные и обучающие метки в качестве первых двух аргументов метода.

Затем могут быть предоставлены validation_data, которые являются нашим тестовым разделением. В большинстве случаев, например, когда вы настраиваете гиперпараметры или выбираете архитектуру модели, вам нужно, чтобы ваш проверочный набор был истинным проверочным набором, а не данными для тестирования. В этом случае мы просто демонстрируем, как обучать нейронную сеть с нуля с помощью Keras, поэтому мы не используем однотипных рекомендаций. Последующие главы этой книги, а также более продвинутое содержание в пакетах Practitioner Bundle и ImageNet Bundle гораздо более строги в отношении научного метода; однако пока просто сосредоточьтесь на коде и поймите, как обучается сеть.

Мы проводим нашу сеть обучаясь общими сложностями 100 эпох, используя размер пакета 128 точек данных за раз. Метод возврата словаря H, который мы будем использовать для построения графика потерь/точности сети в течение нескольких блоков кода.

Как только сеть завершит обучение, мы можем оценить ее на данных тестирования чтобы получить нашу окончательную классификацию

```

50 # оценить сеть
51 print("[ИНФ О оценка сети...") 52 предсказания=
model.predict(testX, batch_size=128) 53 print(classification_report(testY.argmax(axis=1),
54     пр огнозыargmax(ось=1),
55     target_names=[str(x) для x в lb.classes_]))
```

В первом методе модели .predict вернет вероятности меток класса для каждой точки данных в testX (строка 52). Таким образом, если бы вы провели массив прогнозов NumPy, он имел бы форму (X, 10), поскольку в наборе для тестирования всего 17 500 точек данных и десять возможных меток классов (цифры 0-9).

Таким образом, каждая запись в данной строке является вероятностью. Чтобы определить класс с наибольшей вероятностью, мы можем просто вызвать .argmax(axis=1), как в строке 53, чтобы даст нам индекс метки класса с наибольшей вероятностью и, следовательно, нашу окончательную вью одную классификацию. Окончательная классификация вью одних данных посети заносится в таблицу, а затем отчет об окончательной классификации отображается на нашей консоли в строках 53-55.

Наши последний блок кода обрабатывает график потерь при обучении, точности обучения, потерь при проверке и точность проверки в времени:

```

57 # график потерь и точности обучения58
plt.style.use("ggplot") 59 plt.figure() 60
plt.plot(np.arange(0, 100), H.history["loss"], label=
"train_loss") 61 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss") 62
plt.plot(np.arange(0, 100), H.history ["acc"], label="train_acc") 63 plt.plot(np.arange(0,
100), H.history["val_acc"], label="val_acc") 64 plt.title(" Потери при обучении и
Точность") 65 plt.xlabel("Эпох а #") 66 plt.ylabel("Потери/Точность") 67 plt.legend() 68
plt.savefig(args["output"])
```

Затем этот график сохраняется на диск на основе аргумента командной строки --output.

Чтобы обучить нашу сеть полно связных слоев на MNIST, просто выполните следующую команду:

```

$ python keras_mnist.py --output output/keras_mnist.png [INFO]
загружает набор данных MNIST (полный)...
[INFO] тренировочная сеть...
Обучение на 52500 выборках , проверка на 17500
выборках Эпох а 1/100 1 с - потеря 2,2997 - акк: 0,1088 -
val_loss: 2,2918 - val_acc: 0,1145 Эпох а 2/100 1 с - потеря 2,2866 - акк: 0,1133 -
val_loss: 2,2796 - val_acc: val_acc: val_acc: 2,2796 0,1233 Эпох а 3/100 1 с - потеря
2,2721 - акк: 0,1437 - знач_потеря 2,2620 - знач_акк: 0,1962

...
Эпох а 98/100
1 с - потеря 0,2811 - акк: 0,9199 - val_loss: 0,2857 - val_acc: 0,9153 Эпох а 99/100 1
с - потеря 0,2802 - акк: 0,9201 - val_loss: 0,2862 - val_acc: 0,9148 Epoch10 000
```

1 с - убыток: 0,2792 - акк: 0,9204 - знач_убыток: 0,2844 - знач_акк: 0,9160

[INFO] оценка сети...

точность	вспоминать поддержку	f1-score	
0,0	0,94	0,96	0,95
1,0	0,95	0,97	0,96
2,0	0,91	0,89	0,90
3,0	0,91	0,88	0,89
4,0	0,91	0,93	0,92
5,0	0,89	0,86	0,88
6,0	0,92	0,96	0,94
7,0	0,92	0,94	0,93
8,0	0,88	0,88	0,88
9,0	0,90	0,88	0,89
среднее / общее	0,92	0,92	0,92
			17500

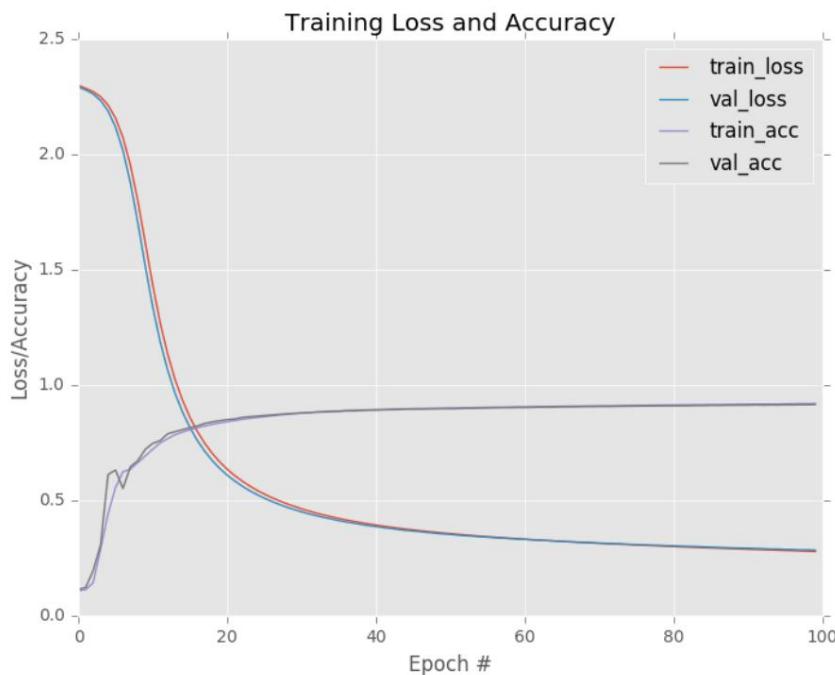


Рисунок 10.14: Обучение нейронной сети прямого распространения 784-256-128-10 с помощью Keras на полном наборе данных MNIST. Обратите внимание, что наши кривые обучения и проверки почти идентичны, что означает наличие не происходит однотипного переобучения

Как показывают результаты мы получаем точность 92%. Кроме того, обучение и проверочное кривые почти идентичны друг другу (рис. 10.14), что указывает на отсутствие переобучения или проблем с тренировочным процессом.

На самом деле, если вы знакомы с набором данных MNIST, вы можете подумать, что точность 92% — это отлично — и это было, пожалуй, 20 лет назад. Как мы узнаем в главе 14, использование Convolutional Нейронные сети, мы можем легко получить > 98% точности. Современные современные подходы могут даже ломать точность 99%.

Хотя на первый взгляд может показаться что наша (строго) полно связная сеть работает хорошо, на самом деле мы можем работать намного лучше. И, как мы видим в следующем разделе, строго полно связные сети, применяемые к более сложным наборам данных, в некоторых случаях могут работать чуть лучше, чем случайные предположения.

СИФ АР-10

Когда речь идет о компьютерном зрении и машинном обучении, набор данных MNIST является классическим определением «контрольного» набора данных, который слишком прост для получения высокоточных результатов и не представляет интереса для изображений, которые мы видим в реальном мире. Мир.

Для более сложного эталонного набора данных мы будем использовать CIFAR-10, набор из 60 000 изображений RGB размером 32×32 , что означает, что каждое изображение в наборе данных представлено $32 \times 32 \times 3 = 3072$ целями числами. Как следует из названия CIFAR-10 состоит из 10 классов, включая самолет, автомобиль, птицу, кошку, оленя, собаку, лягушку, лошадь, корабль и грузовик. Образец набора данных CIFAR-10 для каждого класса можно увидеть на рис. 10.15.

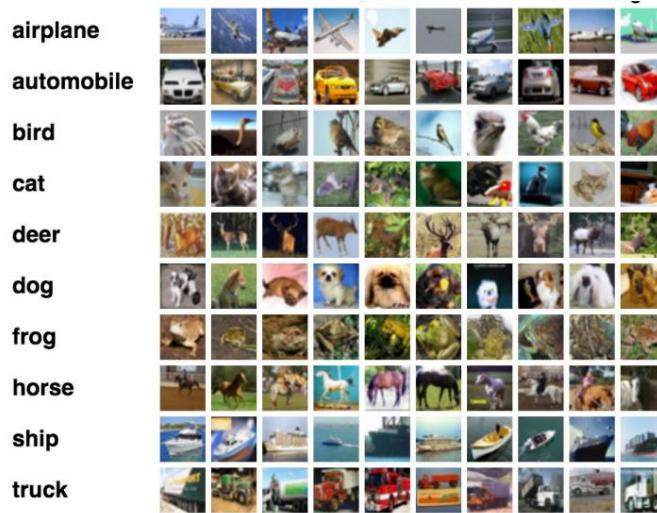


Рисунок 10.15: Примеры изображений из набора данных CIFAR-10 для десяти классов.

Каждый класс равномерно представлен 6000 изображений на класс. При обучении и оценке модели машинного обучения на CIFAR-10 обычно используют предварительно определенные авторами разбиения данных и используют 5000 изображений для обучения и 10 000 для тестирования.

CIFAR-10 значительно сложнее, чем набор данных MNIST. Проблема в том, что изображение, содержащее зеленый пиксель с заданной координатой (x, y) , является лягушкой. Этот пиксель может быть фоном леса, в котором есть олень. Или это может быть цвет зеленого автомобиля или грузовика.

Эти предположения резко контрастируют с набором данных MNIST, где сеть может изучать предположения относительно пространственного распределения интенсивности пикселей. Например, пространственное распределение пикселей переднего плана для 1 существенно отличается от 0 или 5. Этот тип различий во внешнем виде объекта делает наложение серии полностью связных слоев намного более сложным. Как мы знаем из оставшейся части этого раздела, стандартные многоуровневые сети FC (полностью связные) не подходят для такого типа классификации изображений.

Давайте идти вперед и начать. Откройте новый файл, назовите его keras_cifar10.py и вставьте следующий код:

```

1 # импортируем необх одимые пакеты2
из sklearn.preprocessing импортируем LabelBinarizer 3 из
sklearn.metrics импортируем classification_report 4 из keras.models
import Sequential 5 из keras.layers.core импортируем Dense 6 из
keras.optimizers импортируем SGD 7 из keras.datasets
импортируем cifar10 8 импортировать matplotlib.pyplot как plt 9
импортировать numpy как np 10 импортировать argparse

```

Строки 2-10 импортируют наши необх одимые пакеты Python для построения нашей полностью подключенной сети, идентичной предыдущему разделу с MNIST. Исключением является специальная служебная функция в строке 7. Поскольку CIFAR-10 является настолько распространенным набором данных, с которым исследователи сравнивают алгоритмы машинного обучения и глубокого обучения обычно можно увидеть, что библиотеки глубокого обучения предоставляют простые вспомогательные функции для автоматической загрузки этого набора данных с диска.

Далее мы можем проанализировать наши аргументы командной строки:

```

16 # построить разбор аргумента и разобрать аргументы17 ap =
argparse.ArgumentParser() 18 ap.add_argument("-o", "--output",
required=True,
19     help="путь к графику вък одных потер/точности") 20
args = vars(ap.parse_args())

```

Единственный аргумент командной строки, который нам нужен это --output, путь к нашему вък одному графику потер/точности .

Давайте продолжим и загрузим набор данных CIFAR-10:

```

18 # загрузить данные обучения и тестирования, масштабировать их в диапазоне [0, 1],
19 # затем изменить форму матрицы проекта 20 print("[INFO] loading data CIFAR-10...") 21
((trainX, trainY), (testX, testY)) = cifar10.load_data() 22 trainX = trainX.astype("float") / 255.0 23
testX = testX.astype("float") / 255.0 24 trainX = trainX.reshape((trainX.shape[0], 3072)) 25 testX =
testX.reshape((testX.shape[0], 3072))

```

В вов cifar10.load_data в строке 21 автоматически загружает набор данных CIFAR-10 с диска, предварительно сегментированный на обучение и тестирование. Если вы впервые ввываете cifar10.load_data, эта функция извлечет и загрузит набор данных для вас. Этот файл весит 170 МБ, так что наберитесь терпения пока он скачивается и распаковывается. После того, как файл будет загружен один раз, он будет кэширован локально на вашем компьютере, и его не нужно будет загружать снова.

Строки 22 и 23 преобразуют тип данных CIFAR-10 из 8-битных целых чисел без знака в числа с плавающей запятой с последующим масштабированием данных до диапазона [0,1]. Строки 24 и 25 отвечают за изменение формы матрицы проекта для данных обучения и тестирования. Напомним, что каждое изображение в наборе данных CIFAR-10 представлено изображением 32x32x3.

Например, trainX имеет форму (50000, 32, 32, 3), а testX имеет форму (10000, 32, 32, 3). Если бы мы объединили это изображение в один список значений с плавающей запятой, в списке было бы общая сложность $32 \times 32 \times 3 = 3072$ элемента.

Чтобы гладить каждое изображение в тренировочных и тестовых наборах, мы просто используем функцию `.reshape` NumPy. После выполнения этой функции, `trainX` теперь имеет форму (50000, 3072), а `testX` имеет форму (10000, 3072).

Теперь, когда набор данных CIFAR-10 загружен с диска, давайте еще раз бинаризируем класс. цельные числа меток в векторы с последующей инициализацией списка фактических имен меток классов:

```
27 # преобразовать метки из целых чисел в векторы
lb = LabelBinarizer()
28 trainY = lb.fit_transform(trainY)
29 testY = lb.transform(testY)
```

```
31
32 # инициализируем имена меток для набора данных CIFAR-10
33 labelNames = ["самолет", "автомобиль", "птица", "кошка", "оленя", "собака",
34     "лягушка", "лошадь", "корабль", "грузовик"]
```

Пришло время определить архитектуру сети:

```
36 # определить архитектуру 3072-1024-512-10 с помощью Keras
37     _ _ 512, активация="релю") 40 model.add(Dense(10, активация
38     = "softmax"))
```

Строка 37 создает экземпляр класса `Sequential`. Затем мы добавляем первый плотный слой, который имеет `input_shape` 3072, узел для каждого из 3072 сглаженных значений пикселей в матрице дизайна — этот слой затем отвечает за изучение 1024 весов. Мы также заменим устаревший сигмоид на активацию ReLU в надежде улучшить производительность сети.

Следующий полностью связанный слой (строка 39) изучает 512 весов, а последний слой (строка 40) изучает веса, соответствующие десяти возможным видам одноклассификации, а также классификатор `softmax` для получения окончательных в выходных вероятностей для каждого класса.

Теперь, когда архитектура сети определена, мы можем ее обучить:

```
42 # обучить модель с помощью
SGD 43 print("[INFO] training network...")
sgd = SGD(0.01)
44
model.compile(loss="categorical_crossentropy", optimizer=sgd,
45     метрики=["точность"])
46
47 H = model.fit(trainX, trainY, validation_data=(testX, testY),
48     эпох и = 100, размер партии = 32)
```

Мы будем использовать оптимизатор `SGD` для обучения сети со скоростью обучения 0,01, довольно стандартной. первоначальный выбор. Сеть будет обучаться общим сложности 100 эпох, используя партии из 32.

После того, как сеть обучена, мы можем оценить ее с помощью `classification_report`, чтобы получить более подробный обзор производительности модели:

```
50 # оценить сеть
51 print("[INFO] оценка сети...")
52 предсказания =
model.predict(testX, batch_size=32)
53 print(classification_report(testY.argmax(axis=1),
54     прогнозы.argmax(ось=1), target_names=labelNames))
```

И, наконец, мы также построим график потерь/точности с течением времени:

```

56 # график потери и точности обучения
57 plt.style.use("ggplot")
58 plt.figure()
59 plt.plot(np.arange(0, 100), H.history["потеря"], label="train_loss")
60 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
61 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
62 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
63 plt.title("Тренировочные потери и точность")
64 plt.xlabel("Эпох а #")
65 plt.ylabel("Потери/точность")
66 plt.legend()
67 plt.savefig(аргумент["вывод"])

```

Чтобы обучить нашу сеть на CIFAR-10, откройте терминал и выполните следующую команду:

```
$ python keras_cifar10.py --output output/keras_cifar10.png
[INFO] тренировочная сеть...
Обучение на 50 000 образцов, проверка на 10 000 образцах
Эпох а 1/100
7s - убыток: 1,8409 - акк: 0,3428 - val_loss: 1,6965 - val_acc: 0,4070
Эпох а 2/100
7s - убыток: 1,6537 - акк: 0,4160 - val_loss: 1,6561 - val_acc: 0,4163
Эпох а 3/100
7s - убыток: 1,5701 - акк: 0,4449 - val_loss: 1,6049 - val_acc: 0,4376
...
Эпох а 98/100
7s - убыток: 0,0292 - акк: 0,9969 - val_loss: 2,2477 - val_acc: 0,5712
Эпох а 99/100
7s - убыток: 0,0272 - акк: 0,9972 - val_loss: 2,2514 - val_acc: 0,5717
Эпох а 100/100
7s - убыток: 0,0252 - акк: 0,9976 - val_loss: 2,2492 - val_acc: 0,5739
[INFO] оценка сети...
```

	точность	вспоминать поддержку	f1-score	
самолет	0,63	0,66	0,64	1000
автомобиль	0,69	0,65	0,67	1000
птица	0,48	0,43	0,45	1000
Кот	0,40	0,38	0,39	1000
олень	0,52	0,51	0,51	1000
	0,48	0,47	0,48	1000
собака	0,64	0,63	0,64	1000
лягушка лошадь	0,63	0,62	0,63	1000
корабль	0,64	0,74	0,69	1000
грузовик	0,59	0,65	0,62	1000
среднее / общее	0,57	0,57	0,57	10000

Глядя на результат, вы можете видеть, что наша сеть получила точность 57%. Изучение нашего графика потерь и точности с течением времени (рис. 10.16), мы видим, что наша сеть борется с переоснащением прошлой эпохи 10. Потери сначала начинают уменьшаться немноговременно, а затем резко увеличиваются и больше никогда не спускаются. В это время потери на тренировках последовательно снижаются от эпохи к эпохе.

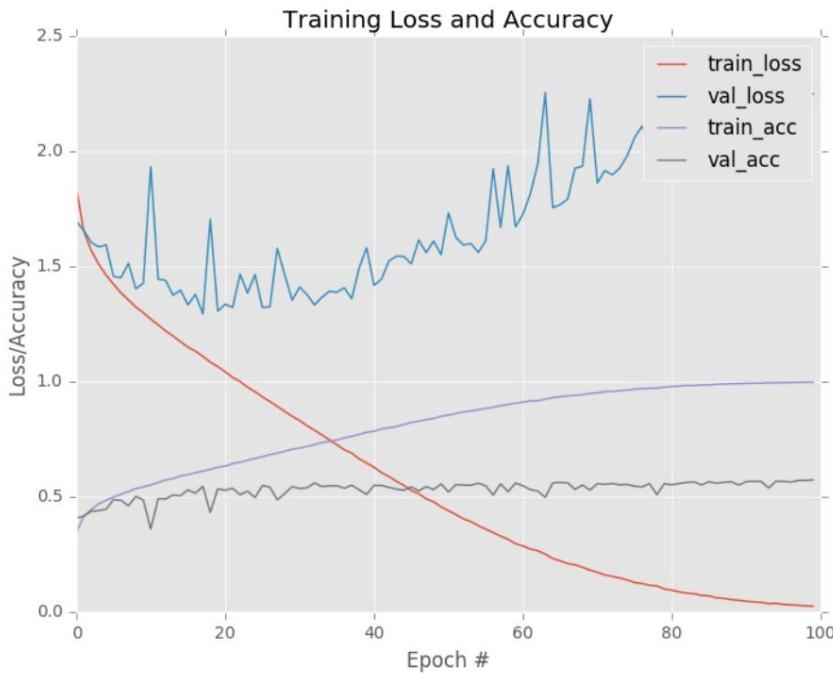


Рисунок 10.16: Использование стандартной нейронной сети с прямой связью приводит к резкому переоснащению в более сложном наборе данных CIFAR-10 (обратите внимание, как потери при обучении падают, а потери при проверке резко возрастают). Чтобы успешно справиться с задачей CIFAR-10, нам понадобится мощная техника — сверточные нейронные сети.

Такое поведение уменьшения потерь при обучении при увеличении потерь при проверке свидетельствует очевидно о переоснащении.

Конечно, мы могли бы рассмотреть возможность дальнейшей оптимизации наших гиперпараметров, в частности, поэкспериментировать с различной скоростью обучения и увеличить как глубину, так и количество узлов в сети, но мы будем бороться за скучные выгоды.

Делов том, что базовые сети прямой связи со строгим полносвязными слоями не подходят для сложных наборов данных изображений. Для этого нам нужен более продвинутый подход: сверточные нейронные сети. К счастью, CNN являются темой всей оставшейся части этой книги. К тому времени, когда вы законите работу со стартовым набором, вы сможете получить более 79% точности на CIFAR-10. Если вы решите более глубоко изучить глубокое обучение, пакет Practitioner Bundle продемонстрирует, как повысить вашу точность до более чем 93%, что поставит нас в лигу самых современных результатов [110].

10.1.5 Четыре компонента рецепта нейронной сети

Возможно, вы начали замечать закономерность в наших примерах кода Python при обучении нейронных сетей. Чтобы собрать собственную нейронную сеть и алгоритм глубокого обучения, вам понадобятся четыре основных компонента: набор данных, модель/архитектура, функция потерь и метод оптимизации. Ниже мы рассмотрим каждый из этих ингредиентов.

Набор данных

Набор данных — это первый компонент обучения нейронной сети — сами данные вместе с проблемой, которую мы пыгаемся решить, определяют наши конечные цели. Например, используем ли мы нейронные сети для выполнения регрессионного анализа, чтобы предсказать стоимость домов в конкретном пригороде через 20 лет? Наша цель

в выполнять неконтролируемое обучение, такое как уменьшение размерности? Или мы пытаемся выполнить классификацию?

В контексте этой книги мы строго придерживаемся классификации изображений; однако сочетание вашего набора данных и проблемы, которую вы пытаетесь решить, влияет на ваш выбор функции потерь, архитектуры сети и метода оптимизации, используемого для обучения модели. Обычно у нас нет выбора в нашем наборе данных (если вы не работаете над собственным проектом) — нам дается набор данных с некоторыми ожиданиями относительно того, какими должны быть результаты нашего проекта. Затем мы должны обучить модель машинного обучения на наборе данных, чтобы она хорошо справлялась с поставленной задачей.

Функция потери

Учитывая наш набор данных и целевую цель, нам нужно определить функцию потерь, которая соответствует проблеме, которую мы пытаемся решить. Почти во всех задачах классификации изображений, использующих глубокое обучение, мы будем использовать кросс-энтропийную потерю. Для 2 классов мы называем это категориальной кросс-энтропией. Для двух классовых задач мы называем бинарную потерю кросс-энтропией.

Модель/Архитектура

Ваша сетевая архитектура может считаться первым фактическим «в выбором», который вы должны сделать в качестве ингредиента. Ваш набор данных, скорее всего, в выбрана в вас (или, по крайней мере, вырешили, что хотите работать с данным набором данных). И если вы выполняете классификацию, вы повсей вероятности, будете использовать кросс-энтропию в качестве функции потерь.

Однако архитектура вашей сети может существенно различаться особенно в зависимости от того, какой метод оптимизации вы выберете для обучения своей сети. Потратив время на изучение вашего набора данных и посмотрите на:

1. Сколько точек данных у вас есть.
2. Количество занятий.
3. Насколько похожи/различны классы
4. Внутри классовая дисперсия

Вы должны начать «чувствовать» сетевую архитектуру, которую собираетесь использовать. Этотребует практики, поскольку глубокое обучение — это отчасти наука, отчасти искусство — на самом деле остальная часть этой книги посвящена тому, чтобы помочь вам развить оба этих навыка.

Имейте в виду, что количество слоев и узлов в вашей сетевой архитектуре (наряду с любым типом регуляризации), вероятно, изменится по мере того, как вы будете проводить все больше и больше экспериментов. Чем больше результатов вы добьетесь, тем лучше вы будете подготовлены принять обоснованные решения от том, какие технологии использовать дальше.

Метод оптимизации

Последним компонентом является определение метода оптимизации. Как мы уже видели в этой книге, стоящий альгоритм градиентный спуск (раздел 9.2) используется довольно часто. Существуют и другие методы оптимизации, включая RMSprop [90], Adagrad [111], Adadelta [112] и Adam [113]; однако это более продвинутые методы оптимизации, которые мы рассмотрим в Практическом пакете.

Даже несмотря на все эти новые методы оптимизации, SGD по-прежнему остается рабочей лошадкой глубокого обучения — большинство нейронных сетей обучаются помощью SGD, в том числе сети, получающие современную точность на сложных наборах данных изображений, таких как ImageNet.

При обучении сетей глубокого обучения особенно когда вы только начинаете изучать ее, SGD должен быть вашим оптимизатором в выборе. Затем вам нужно установить правильную скорость обучения и силу регуляризации, общее количество эпох, для которых должна быть обучена сеть, и следует ли использовать импульс (если да, то какое значение) или ускорение Нестерова. Потратите время, чтобы экспериментировать с SGD как можно больше и освоиться с настройкой параметров.

Знакомство с задачами алгоритмом оптимизации покажет насколько легкие навыки в ожидании.

машина – выездите на своей машине лучше, чем на чужих, потому что провели за рулем так много времени; Вы понимаете свою машину и ее тонкости. Часто данный оптимизатор выбирается для обучения сети на наборе данных не потому, что сам оптимизатор лучше, а потому, что создатель (т. е. специалист по глубокому обучению) лучше знаком с оптимизатором и понимает «искусство» настройки соответствующих параметров.

Имейте в виду, что для получения достаточно производительной нейронной сети даже на небольшом среднем наборе данных может потребоваться от 10 до 100 экспериментов даже для приведения пользователей глубокого обучения — не расстраивайтесь, если ваша сеть не работает очень хорошо с самого начала. . Чтобы научиться глубокому обучению, потребуются затраты времени и множество экспериментов, но оно того стоит, как только вы поймете, как эти ингредиенты сочетаются друг с другом.

10.1.6 Инициализация веса

Прежде чем мы закроем эту главу, я хотел кратко обсудить концепцию инициализации весов, или, проще говоря то, как мы инициализируем наши весовые матрицы векторы смешения.

Этот раздел не предназначен для изучения описания методов инициализации; тем не менее, он выделяет популярные методы из литературы по нейронным сетям и общих практических правил. Чтобы проиллюстрировать, как работают эти методы инициализации веса, являющихся базовым псевдокодом, подобный Python/NumPy, когда это уместно.

10.1.7 Инициализация констант

При применении постоянной нормализации все веса в нейронной сети инициализируются постоянным значением C . Обычно C равно нулю или единице.

Чтобы визуализировать это в псевдокоде, давайте рассмотрим произвольный слой нейронной сети, который имеет 64 входа и 32 выхода (исключая любое смешение для удобства). Чтобы инициализировать эти веса с помощью NumPy и нулевой инициализации (по умолчанию используется Caffe, популярной платформой глубокого обучения), мы должны выполнить:

```
>>> W = np.zeros((64, 32))
```

Точно так же одна инициализация может быть выполнена с помощью:

```
>>> W = np.ones((64, 32))
```

Мы можем применить константную инициализацию, используя произвольный C , используя

```
>>> W = np.ones((64, 32)) * C
```

Хотя константную инициализацию легко понять и понять, проблема с использованием этого метода заключается в том, что нам практически невозможно нарушить симметрию активаций [114]. Поэтому он редко используется в качестве инициализатора веса нейронной сети.

10.1.8 Равномерное и нормальное распределения

Равномерное распределение извлекает случайное значение из диапазона [нижний, верхний], где каждое значение внутри этого диапазона имеет равную вероятность быть выбранным.

Опять же, давайте предположим, что для данного слоя в нейронной сети у нас есть 64 входа и 32 выхода. Затем мы хотим инициализировать наши веса в диапазоне нижний = -0,05 и верхний = 0,05.

Применение следующего кода Python + NumPy позволит нам добиться желаемой нормализации:

```
>>> W = np.random.uniform(низкий = -0,05, високий = 0,05, размер = (64, 32))
```

В выполнение приведенного выше кода NumPy случайным образом сгенерирует $64 \times 32 = 2048$ значений из диапазона $[-0,05, 0,05]$, где каждое значение в этом диапазоне имеет равную вероятность.

Затему нас есть нормальное распределение, где мы определяем плотность вероятности для распределения Гаусса как:

$$p(x) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (10.6)$$

Наиболее важными параметрами здесь являются μ (среднее значение) и σ (стандартное отклонение). Квадрат стандартного отклонения σ , называется дисперсией.

При использовании библиотеки Keras класс RandomNormal извлекает случайные значения из нормального распределения с $\mu = 0$ и $\sigma = 0,05$. Мы можем имитировать это поведение, используя NumPy ниже:

```
>>> W = np.random.normal(0,0, 0,5, размер = (64, 32))
```

Для инициализации весов в нейронных сетях можно использовать как равномерное, так и нормальное распределение, однако обычно мы применяем различные эвристики для создания «лучших» схем инициализации (как мы обсудим в оставшихся разделах).

10.1.9 Униформа и норма Лекуна

Если вы когда-либо использовали фреймворки Torch7 или PyTorch, вы могли заметить, что метод инициализации веса по умолчанию называется «Efficient Backprop», который является результатом работы LeCun et al. [17].

```
>>> F_in = 64 >>>
F_out = 32 >>>
limit = np.sqrt(3 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

Мы также можем использовать нормальное распределение. Библиотека Keras использует усеченную нормаль распределения при построении нижнего и верхнего пределов вместе с нулевым средним:

```
>>> F_in = 64 >>>
F_out = 32 >>>
limit = np.sqrt(1 / float(F_in))
>>> W = np.random.normal(0.0, предел, размер=(F_in, F_out))
```

10.1.10 Униформа Глорота/Ксавьера и обычная

Метод инициализации веса по умолчанию, используемый в библиотеке Keras, называется «инициализация Glorot» или «инициализация Xavier» в честь Ксавьера Glorot, первого автора статьи «Понимание сложности обучения нейронных сетей с глубокой прямой связью» [115].

Для нормального распределения предельное значение строится путем усреднения F_{in} и F_{out} вместе, а затем извлечения квадратного корня [116]. Затем используется нулевой центр ($\mu = 0$):

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in + F_out))
>>> W = np.random.normal(0.0, предел, размер=(F_in, F_out))
```

Инициализация Glorot/Xavier также может быть выполнена с однородным распределением, где мы размещаем более сильные ограничения на лимит:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in + F_out))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

Обучение с использованием этого метода инициализации, как правило, довольно эффективно, и я рекомендую его для большинства нейронных сетей.

10.1.11 He et al./Kaiming/MSRA Uniform and Normal

Часто упоминается как «He et al. инициализация», «инициализация Kaiming» или просто «инициализация MSRA», этот метод назван в честь Kaiming He, первого автора статьи «Углубление в вспомогательные превосходства на уровне человека в классификации ImageNet» [117].

Обычно мы используем этот метод при обучении очень глубоких нейронных сетей, используя широкие ReLU-подобные функции активации (в частности, «PReLU» или Parametric Rectified Linear Unit).

Чтобы инициализировать веса в слое с помощью He et al. инициализация с равномерным распределением мы установите `limit` равным $\text{limit} = \sqrt{6/F_{\text{in}}}$, где F_{in} — количество входов единиц в слое:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(6 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

Мы также можем использовать нормальное распределение, установив $\mu = 0$ и $\sigma = \sqrt{2/F_{\text{in}}}$

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(2 / float(F_in))
>>> W = np.random.normal(0.0, предел, размер=(F_in, F_out))
```

Мы обсудим этот метод инициализации как в Practitioner Bundle, так и в ImageNet.

Набор этой книги, в которой мы обучаем очень глубокие нейронные сети на больших наборах данных изображений.

10.1.12 Различия в реализации инициализации

Фактические предельные значения могут отличаться для LeCun Uniform/Normal, Xavier Uniform/Normal и He et al. Равномерное/нормальное. Например, при использовании Xavier Uniform в Caffe $\text{limit} = -\sqrt{3/F_{\text{in}}}$ [114]; однако при инициализации Xavier по умолчанию для Keras используется $\sqrt{6/(F_{\text{in}} + F_{\text{out}})}$ [118]. Ни один метод не является «более правильным», чем другой, но вам следует прочитать документацию соответствующей библиотеки глубокого обучения.

10.2 Резюме

В этой главе мы рассмотрели основы нейронных сетей. В частности, мы сосредоточились на истории нейронных сетей и их связи с биологией.

Оттуда мы перешли к искусственным нейронным сетям, таким как алгоритм Perceptron. Хотя алгоритм Perceptron важен с исторической точки зрения он имеет один существенный недостаток — он точно классифицирует нелинейные разделяемые точки. Для работы с более сложными наборами данных нам нужны как (1) нелинейные функции активации, так и (2) многослойные сети.

Для обучения многослойных сетей мы должны использовать алгоритм обратного распространения ошибки. Затем мы вручную реализовали обратное распространение и продемонстрировали, что при использовании для обучения многослойных сетей с нелинейными функциями активации мы можем моделировать нелинейно разделяемые наборы данных, такие как XOR.

Конечно, реализация обратного распространения вручную — сложный процесс, подверженный ошибкам — поэтому мы часто полагаемся на существующие библиотеки, такие как Keras, Theano, TensorFlow и т. д. Это позволяет нам сосредоточиться на фактической архитектуре, а не на базовом алгоритме, используемом для обучать сеть.

Наконец, мы рассмотрели четыре ключевых компонента при работе с любой нейронной сетью, включая набор данных, функция потерь, модель/архитектура и метод оптимизации.

К сожалению, как показали некоторые из наших результатов (например, CIFAR-10), стандартные нейронные сети не могут обеспечить высокую точность классификации при работе с сложными наборами данных изображений, которые демонстрируют различия в переводе, вращении, точке обзора и т. д. Чтобы получить разумную точность на этих наборах данных, нам нужно будет работать со специальным типом нейронных сетей с прямой связью, называемых сверточными нейронными сетями (CNN), которые как раз являются предметом нашей следующей главы.

11. Сверточные нейронные сети

Весь наш обзор машинного обучения и нейронных сетей до сих пор сводился к этому моменту: пониманию сверточных нейронных сетей (CNN) и роли, которую они играют в глубоком обучении.

В традиционных нейронных сетях с прямой связью (подобных тем, которые мы изучали в главе 10) каждый нейрон в одном слое связан с каждым в *всем* другим нейроном в следующем слое — мы называем это полносвязанным (FC) слоем. Однако в CNN мы не используем уровень FC до самого последнего уровня (слоев) в сети. Таким образом, мы можем определить CNN как нейронную сеть, которая заменяет специализированный «сверточный» слой в месте «полностью связанного» слоя по крайней мере для одного из слоев в сети [10].

Затем в *всех* однотипных этих сверточных применяется нелинейная функция активации, такая как ReLU, и процесс свертки => активация продолжается (вместе с остальными другими типами слоев, чтобы помочь уменьшить ширину и высоту в *всем* объеме и помочь уменьшить переоснащение), пока мы не дойдем до конца сети и не применим один или два слоя FC, где мы можем получить наши окончательные в *всем* однотипные классификации.

Каждый уровень в CNN применяет свой набор фильтров, общую сотни или тысячи фильтров, и объединяет результаты передавая в *всем* однотипные данные на следующий уровень в сети. В время обучения CNN автоматически изучает значения этих фильтров.

В контексте классификации изображений наша CNN может научиться •

Обнаруживать края из необработанных данных пикселей в первом слое. •

Используйте эти ребра для обнаружения форм (т. е. «капель») во втором слое. •

Используйте эти формы для обнаружения элементов более высокого уровня, таких как структуры лица, части автомобиля и т. д. в верхних слоях сети.

Последний уровень в CNN использует эти функции более высокого уровня для прогнозирования содержимого изображения. На практике CNN дают нам два ключевых преимущества: локальную инвариантность и композиционность. Концепция локальной инвариантности позволяет нам классифицировать изображение как содержащее определенный объект независимо от того, где на изображении появляется этот объект. Мы получаем эту локальную инвариантность за счет использования «единичных слоев» (обсуждаемых далее в этой главе), которые определяют области нашего в *всем* объеме с высокой реакцией на определенный фильтр.

Второе преимущество — композиционность. Каждый фильтр составляет локальный патч функций более низкого уровня

в представление более высокого уровня аналогично тому, как мы можем составить набор математических функций, основанных на вводах предыдущих функций: $f(g(x(h(x)))$ — эта композиция позволяет нашей сети изучать более богатые функции глубже в сети. Например, наша сеть может создавать ребра из пикселей, фигуры из ребер, а затем сложные объекты из фигур — все это происходит автоматически, что происходит естественным образом в процессе обучения. Концепция построения признаков более высокого уровня из более низких. Именно поэтому CNN так сильны в компьютерном зрении.

В оставшейся части этой главы мы точно обсудим, что такое свертки и какую роль они играют в глубоком обучении. Затем мы перейдем к строительным блокам CNN: слоям различного типа слоев, которые вы будете использовать для создания собственных CNN. Мы завершим эту главу, рассмотрев общие шаблоны, которые используются для объединения этих строительных блоков для создания архитектур CNN, которые хорошо справляются с разнообразным набором задач классификации изображений.

Изучив эту главу, мы будем иметь (1) четкое представление о сверточных нейронных сетях и мыслительном процессе, который входит в их создание, и (2) ряд «рецептов» CNN, которые мы можем использовать для построения наших собственных сетевых архитектур. В нашей следующей главе мы будем использовать эти основные рецепты для обучения собственных CNN.

11.1 Понимание сверток

В этом разделе мы рассмотрим ряд вопросов, в том числе:

- Что такое свертки изображений?
- Что они делают? • Почему мы их используем?
- Как мы применяем их к изображениям?
- Какую роль играют свертки в глубоком обучении?

Слово «свертка» звучит как причудливый, сложный термин, но на самом деле это не так. Если у вас уже есть опыт работы с компьютерным зрением, обработкой изображений или OpenCV, вы уже применяли свертки, осознавая это или нет!

Вы когда-нибудь применяли размытие или сглаживание к изображению? Да, это свертка. Как насчет обнаружения краев? Ага, свертка. Вы открывали Photoshop или GIMP, чтобы повысить резкость изображения? Выгадали - свертка. Свертки являются одним из наиболее важных фундаментальных строительных блоков в компьютерном зрении и обработке изображений.

Но сам термин имеет тенденцию отпугивать людей — на самом деле, на первый взгляд, это слово даже кажется имющим пренебрежительный оттенок (зачем кому-то отдать что-то «скручивать»?) Поверьте мне, свертки — это совсем не страшно. На самом деле их довольно легко понять.

Сточки зрения глубокого обучения свертка (изображения) — это поэлементное умножение двух матриц, трижды за которыми следует сумма.

Шутки в сторону. Вот и все. Вы только что узнали, что такое свертка: 1. Возьмите две матрицы (обе имеют одинаковые размеры).
2. Умножьте их поэлементно (т.е. не скалярным произведением, а простым умножением).
3. Сложите элементы в месте.

Мы узнаем больше о свертках, ядрах и отом, как они используются внутри CNN, в оставшейся части этого раздела.

11.1.1. Свертки и взаимная корреляция

Читатель, ранее знакомый с компьютерным зрением и обработкой изображений, возможно, идентифицировал мое описание свертки выше как операцию взаимной корреляции. Использование кросс-корреляции вместо свертки на самом деле задумано. Свертка (обозначается оператором \star) над

двумерное в x одно изображение I и двумерное ядро K определяется как:

$$S(i, j) = (I * K)(i, j) = \sum_{m=-n}^{n} \sum_{j=-n}^{n} K(i-m, j-n)I(m, n) \quad (11.1)$$

Однако почти все библиотеки машинного обучения и глубокого обучения используют упрощенный кросс-курс. корреляционная функция

$$S(i, j) = (I * K)(i, j) = \sum_{m=-n}^{n} \sum_{j=-n}^{n} K(i+m, j+n)I(m, n) \quad (11.2)$$

Вся эта математика сводится к изменению знака в том, как мы получаем доступ к координатам изображения I (т.е. нам не нужно «переворачивать» ядро относительно двух данных при применении кросс-корреляции).

Отметь же, многие библиотеки глубокого обучения используют упрощенную операцию кросс-корреляции и называют ее сверткой — здесь мы будем использовать ту же терминологию. Для читателей, заинтересованных в получении дополнительной информации о математике, лежащей в основе свертки и взаимной корреляции, обратитесь к главе 3 книги «Компьютерное зрение: алгоритмы приложения» Шельского [119].

11.1.2 Аналогия «большой матрицы» и «окраинной матрицы»

Изображение представляет собой многомерную матрицу. Наше изображение имеет ширину (количество столбцов) и высоту (количество строк), как и матрица. Но в отличие от традиционных матриц, с которыми вы работали еще в начальной школе, у изображений также есть глубина — количество каналов в изображении.

Для стандартного изображения RGB у нас есть глубина 3 — по одному каналу для каждого из красного, зеленого и синего каналов соответственно. Учитывая это значение, мы можем думать об изображении как об большой матрице, а ядре или сверточной матрице — как окраинной матрице, которая используется для размытия повышения резкости, обнаружения краев и других функций обработки. Последнее, это окраинное ядро на которых сводится к каждому (x, y)-координате исходного изображения.

Это нормально — определять ядра вручную для получения различных функций обработки изображений. На самом деле вы можете уже знакомы с размытием (усредненное сглаживание, сглаживание по Гауссу, срединное сглаживание и т. д.), обнаружением краев (лапласиан, Собель, Шарр, Превитт и т. д.) и повышением резкости — все эти операции являются формами ручных операций. Определение ядра, специально предназначенное для выполнения определенной функции.

Поэтому возникает вопрос: есть ли способ автоматически изучить эти типы фильтров? И даже использовать эти фильтры для классификации изображений и обнаружения объектов? Выдержите пару, что есть. Но прежде чем мы это сделаем, нам нужно немного больше понять ядра и свертки.

11.1.3 Ядра

Отметь же, давайте подумаем об изображении как об большой матрице, а ядре — как окраинной матрице (по крайней мере, по отношению к исходному изображению «большой матрицы»), изображенной на рис. 11.1. Как показано на рисунке, мы перемещаем ядро (красная область) слева направо и сверху вниз по исходному изображению. В каждой (x, y)-координате исходного изображения мы ставим ядро и исследуем окрестности пикселей, расположенных в центре ядра изображения. Затем мы берем эту окрестность пикселей, складываем их с ядром и получаем одно значение. В этом значение сохраняется в том же изображении с теми же (x, y)-координатами, что и центр ядра.

Если это звучит запутанно, не беспокойтесь, мы рассмотрим пример в следующем разделе. Но прежде чем мы углубимся в пример, давайте посмотрим, как выглядит ядро (рис. 11.3):

$$K = \frac{1}{9} \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} \quad (11.3)$$

131	162	232	84	91	207
104	-1	109	+1	237	109
243	-2	202	+2	135	126
185	-1	20	+1	61	225
157	124	25	14	102	108
5	155	16	218	232	249

Рис. 11.1. Ядро можно представить в виде небольшой матрицы, которая скользит слева направо и сверху вниз по большему изображению. В каждом пикселе в х одногоЗображения окрестность изображения свертывается ядром, а в х одногоЗображения сохраняются

В ще мы определили квадратное ядро 3×3 (есть предположения для чего используется это ядро?). Ядра могут иметь произвольный прямоугольный размер $M \times N$ при условии, что M и N являются нечетными целыми числами.

 Большинство ядер, применяемых для глубокого обучения CNN, представляют собой квадратные матрицы размера $N \times N$, что позволяет нам использовать преимущества оптимизированной библиотек линейной алгебры, которые наиболее эффективно работают с квадратными матрицами.

Мы используем нечетный размер ядра, чтобы убедиться, что в центре изображения есть допустимая целочисленная координата (x, y) (рис. 11.2). Слева у нас есть матрица 3×3 . Центр матрицы расположен в точках $x = 1, y = 1$, где в верхний левый угол матрицы используется качестве начала координат, а наши координаты имеют нулевой индекс. Но справа у нас есть матрица 2×2 . Центр этой матрицы будет расположен в точках $x = 0,5, y = 0,5$.

Но, как мы знаем, без применения интерполяции не существует такого понятия, как расположение пикселя $(0,5,0,5)$ — наши координаты пикселя должны быть целыми числами! Именно поэтому мы используем нечетные размеры ядра: чтобы всегда гарантировать наличие допустимой (x, y) -координаты в центре ядра.

11.1.4 Пример ручного выполнения свертки

Теперь, когда мы обсудили основы ядер, давайте обсудим фактическую операцию свертки и посмотрим на пример ее фактического применения, чтобы помочь нам закрепить наши знания. При обработке изображений свертка требует трех компонентов:

1. В x одногоЗображение.
2. Матрица ядра, которую мы собираемся применить к x одногоЗображению.

131	162	232
104	93	139
243	26	252

131	162
104	?

Ниже вы можете найти пример свертки (математически обозначается как оператор ?) а

ох, дж —

=

Вот и все! Свертка - это просто сумма поэлементного матричного умножения

```

1 # импортируем необх одимые пакеты
2 из skimage.exposure import rescale_intensity
3 импортировать numpy как np
4 импортировать сингаксический анализ
5 импорт cv2

```

Мы начиная с строк 2-5 , импортируя необх одимые пакеты Python. Мы будем использовать NumPy и OpenCV для нашей стандартной обработки числовых массивов и функций компьютерного зрения, а также библиотеку scikit-image, которая поможет нам реализовать нашу собственную функцию свертки.

Далее мы можем приступить к определению этого метода свертки:

7 **дeф свернуть (изображение, K):**

```

8     # получаем пространственные размеры изображения ядра
9     (iH, iW) = изображение.форма[:2]
10    (kH, kW) = K.shape[:2]
11
12    # выделить память для вьюк одного изображения, позаботившись о "дополнении"
13    # границах одного изображения, поэтому пространственный размер (т.е.
14    # ширину и высоту) не уменьшаются
15    pad = (kW - 1) // 2
16    image = cv2.copyMakeBorder(изображение, накладка, накладка, накладка, накладка,
17                           cv2.BORDER_REPLICATE)
18    вьюод = np.zeros((iH, iW), dtype="float")

```

Для функции свертки требуется два параметра: изображение (оттенки серого) , которое мы отнимем свертывая ядром. Учитывая как наш образ , так и ядро (которое, как мы предполагаем, является NumPy массивом) , затем мы определяем пространственные размеры (т. е. ширину и высоту) каждого (строки 10 и 11).

Прежде чем мы продолжим, важно понять процесс «скользжения» сверточной матрицы по изображению , применение свертки , а затем соединение ввода, который на самом деле будет уменьшить пространственные размеры на всех одних изображения . Почему это?

Напомним, что мы «центрируем» наши вычисления вокруг центра (x, y)-координат в одних изображения над которыми в данный момент нас одит ядро. Такое расположение подразумевает, что такого понятия как «центр», не существует. Пиксели для пикселей, которые падают в даль границы изображения (поскольку углы ядра будут «зависание» изображения где значения не определены) , изображенное на рис. 11.3.

Уменьшение пространственного размера — это простой побочный эффект применения сверток к изображениям. Иногда этот эффект желателен, а иногда нет, это просто зависит от вашего приложения.

Однако в большинстве случаев мы отнимем, чтобы наше вьюк одно изображение имел те же размеры что и наше вьюк одно изображение. Чтобы обеспечить одинаковые размеры мы применяем отступы (строки 15-18). Мы здесь просто копируем пиксели в даль границы изображения чтобы вьюк одно изображение совпадало размерами с одни изображения.

Существуют и другие методы заполнения в том числе заполнение нулями (заполнение границ нулями — очень распространено при построении сверточных нейронных сетей) и обтекание (где граничные пиксели определяются путем изучения противоположной стороны изображения). В большинстве случаев вы видите либо реплицировать или заполнить нулями. Репликационная прокладка чаще используется когда речь идет об эстетике.

В то время как нулевое заполнение лучше всего подходит для эффективности.

Теперь мы готовы применить настоящую свертку к нашему изображению :

```

20    # цикл по вьюк одному изображению, "скользящий" по нему ядро
21    # каждая(x, y)-координата слева направо и сверху вниз

```

-1	0	+1					
-2	01	+22	232	84	91	207	
-1	04	+13	139	101	237	109	
243	26	252	196	135	126		
185	135	230	48	61	225		
157	124	25	14	102	108		
5	155	116	218	232	249		

Рисунок 11.3: Если бы мы попытались применить свертку к пикселью, расположенному в точке (0,0), то наше ядро 3×3 будет «свисать» с края изображения. Обратите внимание, что для первой строки и первый столбец ядра. Из-за этого мы всегда либо (1) начинаем свертку в первой допустимой позиции или (2) применить заполнение нулями (описано далее в этой главе).

```

22     для y в np.arange(pad, iH + pad):
23         для x в np.arange(pad, iW + pad):
24             # извлечь область интереса изображения извлекая
25             # *центр* области текущих (x, y)-координат
26             # Габаритные размеры
27             roi = изображение[y - площадка:y + площадка + 1, x - площадка:x + площадка + 1]
28
29             # въполнить настоящую свертку, взяв
30             # поэлементное умножение между ROI и
31             # ядро, затем суммируем матрицу
32             k = (roi * K).сумма()
33
34             # сохранить свернутое значение в вък однокадр (x, y)-
35             # координата вък одного изображения
36             output[y - pad, x - pad] = k

```

Строки 22 и 23 зацикливаются на нашем изображении, «сдвигая» ядро слева направо и сверху вниз, один пиксель за раз. Стока 27 извлекает область интереса (ROI) из изображения с помощью NumPy. Нарезка массива. ROI будет сосредоточена вокруг текущих (x, y)-координат изображения. ROI также будет иметь тот же размер, что и наше ядро, что очень важно для следующего шага.

Свертка выполняется в строке 32 путем поэлементного умножения между ROI и kernel с последующим суммированием записей в матрице. Затем вък одно значение к сох ранжируется

в вик однмассиве в тех же (x, y)-координатах (относительно вх одного изображения).

Теперь мы можем закончить наш метод свертки:

```

38     # изменить масштаб вик одного изображения чтобы оно находилось в диапазоне [0, 255]
39     вик од = rescale_intensity(вик од, in_range=(0, 255)) вик од = (вик од *
40     255).astype("uint8")
41
42     # вернуть вик одно
43     изображение return output

```

При работе с изображениями мы обычно имеем дело со значениями пикселей, попадающими в диапазон [0, 255]. Однако при применении сверток мы можем легко получить значения, выходящие за пределы этого диапазона. Чтобы вернуть наше вик одно изображение в диапазон [0, 255], мы применяем функцию rescale_intensity scikit-image (строка 39).

Мы также преобразуем наше изображение обратно в 8-битный целочисленный тип данных без знака в строке 40 (ранее вик одно изображение было типом данных с плавающей запятой, чтобы обрабатывать значения пикселей за пределами диапазона [0, 255]). Наконец, вик одно изображение возвращается возвращаясь в функции в строке 43.

Теперь, когда мы определили нашу функцию свертки, давайте перейдем к драйверной части скрипта. Этот раздел нашей программы будет обрабатывать аргументы командной строки, определяя ряд ядер, которые мы собираемся применить к нашему образу, а затем отображать вик одно результаты

```

45 # построить разбор аргумента и разобрать аргументы
46 ap =
argparse.ArgumentParser() 47 ap.add_argument("-i", "-image", required=True,
48             help="путь к вик одному изображению")
49 args = vars(ap.parse_args())

```

Наш скрипт требует только одного аргумента командной строки, -image, который является путем к нашему вик одному изображению. Затем мы можем определить два ядра, используемые для размытия и сглаживания изображения

```

51 # построить средние ядра размытия используемые для сглаживания
изображения 52 smallBlur = np.ones((7, 7), dtype="float") * (1.0 / (7 * 7)) 21),
dtype="float") * (1.0 / (21 * 21))

```

Чтобы убедиться что это ядро выполняет размытие, обратите внимание, что каждая запись в ядре представляет собой среднее значение $1/S$, где S — общее количество записей в матрице. Таким образом, это ядро будет умножать каждый вик одной пиксель на малую долю и брать сумму — это и есть определение среднего.

Затем у нас есть ядро, отвечающее за резкость изображения

```

55 # построить фильтр повышения резкости
56 sharpen = np.array([ [0, -1, 0], [-1, 5, -1], [0,
57     -1, 0] ], dtype="int")
58
59

```

Затем ядро Лапласа использовалось для обнаружения реберных областей:

```

61 # построить ядро Лагласа, используемое для обнаружения реберък
62 # области изображения
63 лагласиан= np.array(
64     [0, 1, 0],
65     [1, -4, 1],
66     [0, 1, 0]), dtype="int")

```

Ядра Собеля можно использовать для обнаружения реберък областей как по оси x , так и по оси y соответственно:

```

68 # построить ядро Собеля по оси X
69 sobelX = np.array(
70     [-1, 0, 1],
71     [-2, 0, 2],
72     [-1, 0, 1]), dtype="int")
73
74 # построить ядро Собеля по оси Y
75 sobelY = np.array(
76     [-1, -2, -1],
77     [0, 0, 0],
78     [1, 2, 1]), dtype="int")

```

И, наконец, мы определяем ядро тиснения

```

80 # построить ядро тиснения
81 тиснение = np.array(
82     [-2, -1, 0],
83     [-1, 1, 1],
84     [0, 1, 2]), dtype="int")

```

Объяснение того, как было сформулировано каждое из этих ядер, въкходит за рамки этой книги, поэтому для пока просто примите, что это ядра, которые были собраны вручную для выполнения заданного операция

Для дальнейшего рассмотрения того, как математически конструируются ядра и доказывается их работоспособность конкретную операцию обработки изображения см. в Szeliksi (глава 3) [119]. Я также рекомендую использовать ядро превосходный инструмент визуализации ядра от Setosa.io [120].

Имея все эти ядра, мы можем объединить их в набор кортежей, называемый «банком ядер»:

```

86 # создать банк ядер, список ядер, которые мы собираемся применить
87 # используя как нашу пользовательскую функцию 'convole', так и 'filter2D' OpenCV
88 # функция
89 ядробанк = (
90     ("маленько_размытие", маленько_размытие),
91     ("большое_размытие", большое_размытие),
92     ("точить", точить),
93     («лагласиан», лагласиан),
94     ("собел_x", собелкс),
95     ("собель_y", собель),
96     ("тиснение", тиснение))

```

Построение этого списка ядер позволяет использовать цикл поними визуализировать их в виде эффективно, как показано в приведенном ниже блоке кода:

```

98 # загрузить исх одног изображение и преобразовать его в оттенки серого
99 изображение = cv2.imread(аргументы["изображение"])
100 серый = cv2.cvtColor(изображение, cv2.COLOR_BGR2GRAY)
101

102 # цикл по ядрам
103 для(kernelName, K) в kernelBank:
104     # применить ядро к изображению в градациях серого, используя оба наших
105     # функции 'convolve' и функции 'filter2D' OpenCV
106     print("[INFO] применение {} ядра".format(kernelName))
107     convolveOutput = свертка(серый, K)
108    opencvOutput = cv2.filter2D(серый, -1, K)

109
110     # показать вьюк одног изображения
111     cv2.imshow("Исх одног", серый)
112     cv2.imshow("{} - свертка".format(kernelName), convolveOutput)
113     cv2.imshow("{} - opencv".format(kernelName), opencvOutput)
114     cv2.waitKey(0)
115     cv2.destroyAllWindows()

```

Строки 99 и 100 загружают исх изображение с диска и преобразуют его в оттенки серого. Операторы свертки могут и применяться RGB или другим многоканальным образом, но для простоты мы будем применять наши фильтры только к изображениям в градациях серого.

Мы начинаем перебирать наш набор ядер в kernelBank в строке 103, а затем применяем текущее ядро к серое изображение в строке 104, вьевав метод convolve нашей функции, определенный ранее в сценарии.

В качестве проверки работоспособности мы также вьеваем cv2.filter2D, который также применяет наше ядро к серому цвету изображение. Функция cv2.filter2D — это гораздо более оптимизированная версия OpenCV нашей свертки. Функция основная причина, по которой я клую оба здесь, заключается в том, что мы должны проверить работоспособность нашей пользовательской реализации.

Наконец, строки 111-115 отображают вьюк одног изображения на нашем экране для каждого типа ядра.

Результаты свертки

Чтобы запустить наш скрипт (и визуализировать вьюк различных операций свертки), просто выполните следующие команды:

```
$ python convolutions.py --image jemma.png
```

Затем вы увидите результат применения ядра smallBlur к вх одному изображению на рис. 11.4. Слева у нас есть исх одног изображение. Затем в центре у нас есть результаты свертки функция A справа результаты cv2.filter2D. Быстрый визуальный осмотр покажет что на вьюк соответствует cv2.filter2D, что указывает на то, что наша функция свертки работает правильно.

Кроме того, наше изображение теперь выглядит «размытым» и «сглаженным» благодаря ядру сглаживания.

Применим большее размытие, результаты которого можно увидеть на рис. 11.5 (вверху слева). На этот раз я опускаю результаты cv2.filter2D для экономии места. Сравнивая результаты с рис. 11.5 с на рис. 11.4 обратите внимание, как по мере увеличения размера ядра усреднения степень размытия вьюк одног изображение также увеличивается.

Мы также можем увеличить резкость нашего изображения (рис. 11.5, вверху посередине) и обнаружить области, похожие на края с помощью оператора Лагласа (вверху справа).

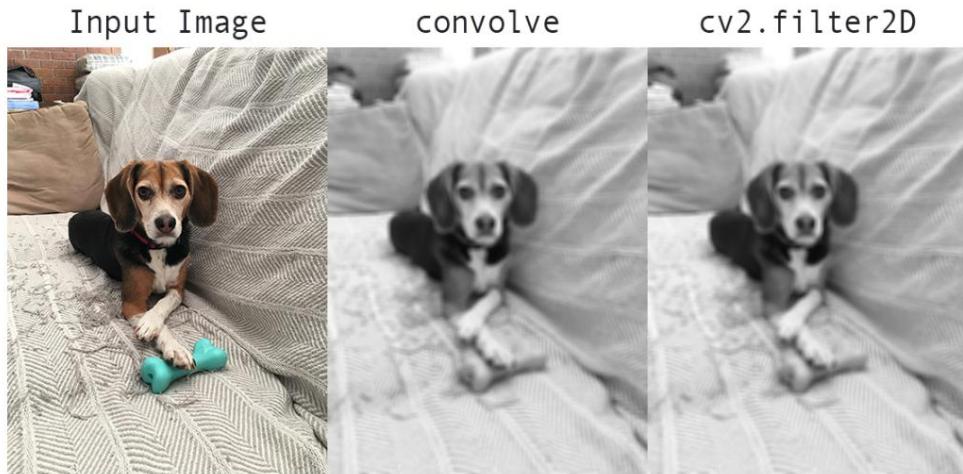


Рисунок 11.4: Слева: исх однє исх однє изображение. В центре: применение среднегоразмытия 7×7 с использованием нашей пользовательской функции свертки. Справа: применение одного и того же размытия 7×7 с использованием cv2.filter2D OpenCV — обратите внимание, что вък однє данье двух функций иденгичны, что означает, что нашметод свертки реализованправильно.

Ядро sobelX используется для поиска вертикальных краев на изображении (рис. 11.5, внизу слева), а ядро sobelY выявляет горизонтальные края (внизу посередине). Наконец, мы можем видеть результат ядра тиснения в левом нижнем углу.

11.1.6 Роль сверток в глубоком обучении

Как вы поняли из этого раздела, мы должны вручную определить каждое из наших ядер для каждой из наших различных операций обработки изображений, таких как сглаживание, повышение резкости и обнаружение краев. Это все хорошо, но что, если бывшеместоэтого был способ выучить эти фильтры?

Можно ли определить алгоритм машинного обучения, который может просматривать наши вък однє изображения в конечном итоге изучать эти типы операторов? На самом деле они есть — именно этим типам алгоритмов посвящена эта книга: сверточные нейронные сети (CNN).

Применяя фильтры свертки, нелинейные функции активации, объединение и обратное распространение, CNN могут изучать фильтры, которые могут обнаруживать ребра и структуры подобные каплям, на более низких уровнях сети, а затем использовать ребра и структуры в качестве «строительных блоков», в конечном итоге обнаруживая объекты высокого уровня (например, лица, кошки, собаки, чаши и т. д.) в более глубоких слоях сети.

Этот процесс использования слоев более низкого уровня для изучения функций высокого уровня есть составная часть CNN, о которой мы говорили ранее. Но как именно CNN это делают? Ответ заключается в целенаправленном укладывании определенного набора слоев. В следующем разделе мы обсудим эти типы слоев, а затем рассмотрим общие шаблоны наложения слоев, которые широко используются в многих задачах классификации изображений.

11.2 Строительные блоки CNN

Как мы узнали из главы 10, нейронные сети принимают вък однє вектор изображения/признака (один вък однє узел для каждой записи) и преобразуют его через ряд скрытых слоев, обычно используя нелинейные функции активации. Каждый скрытый слой также состоит из набора нейронов, где каждый нейрон полностью связан со всеми нейронами предыдущего слоя. Последний слой нейронной сети (то есть «вък однє слой») также является полносвязным и представляет окончательные вък однє классификации сети.

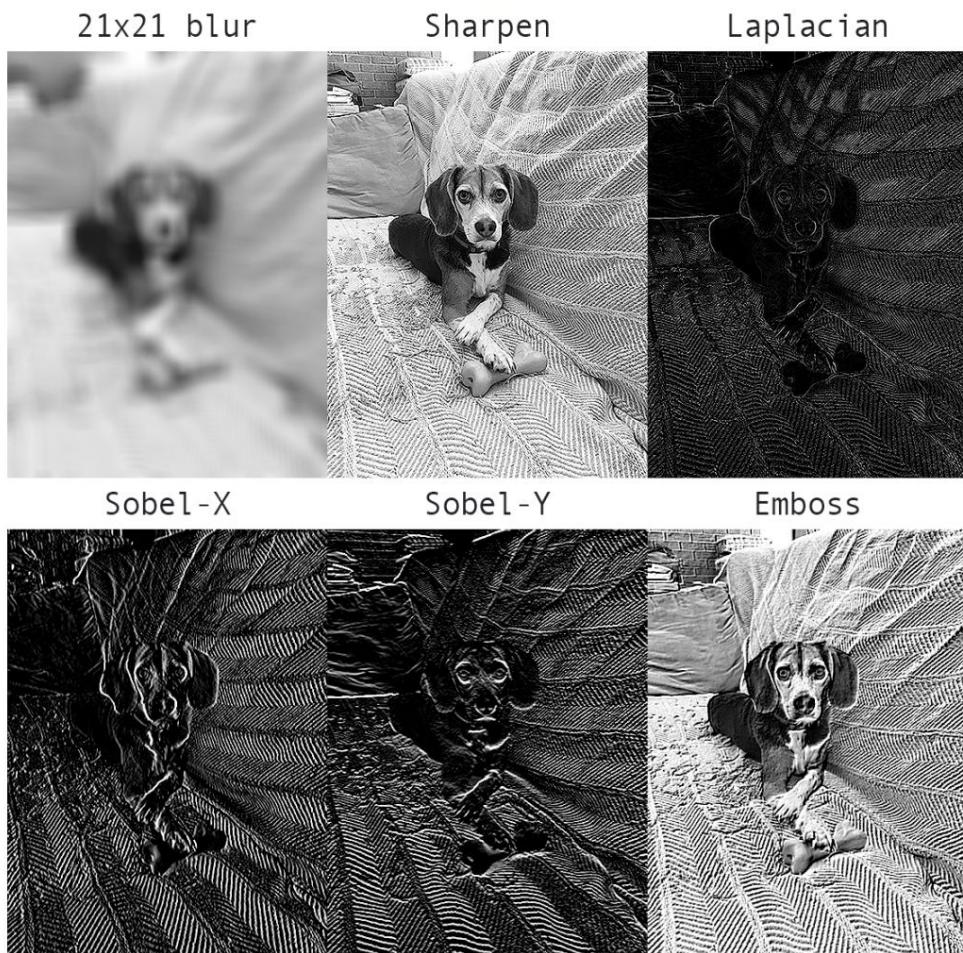


Рисунок 11.5: В верхней левой части: применение среднегоразмыгивания 21×21 . Обратите внимание, что это изображение более размыто, чем на рис. 11.4. В верхней-середине: использование ядра повышения резкости для улучшения деталей. В верхней справа: обнаружение краев в ядре оператора Лапласа. В нижней слева: выявление вертикальных ребер с использованием ядра Sobel-X. Нижняя середина: поиск горизонтальных ребер с использованием ядра Sobel-Y. В нижней справа: применение ядра тиснения.

Однако, как показывают результаты раздела 10.1.4, нейронные сети, работающие непосредственно с интенсивностью необработанных пикселей: 1. Плохо масштабируются при увеличении размера изображения.

2. Оставляют желать большей точности (например, стандартная сеть прямой регистрации на CIFAR-10). получена только 15% точность).

Чтобы продемонстрировать, как стандартные нейронные сети плохо масштабируются при увеличении размера изображения, давайте снова рассмотрим набор данных CIFAR-10. Каждое изображение в CIFAR-10 имеет размер 32×32 с красным, зелеными и синими каналами, что дает в общей сложности $32 \times 32 \times 3 = 3072$ в одни нашу сеть.

Всего 3072 в одни данные не кажется чрезмерно большим, но учтите, что если бы мы использовали изображения размером 250×250 пикселей, общее количество в одни данные и весов подскочило бы до $250 \times 250 \times 3 = 187500$ — и это чисто только для толькx одних слоев! Конечно, мы можем отнести добавить несколько скрытых слоев с различным количеством узлов в слой — эти параметры могут быть складываться и, учитывая низкую производительность стандартных нейронных сетей на необработанных интенсивностях пикселей, это наверняка вряд ли того стоит.

Вместо этого мы можем использовать сверточные нейронные сети (CNN), которые используют в одни данные.

структуру изображения определить сетевую архитектуру более разумным способом. В отличие от стандартной нейронной сети, слои CNN расположены в трехмерном объеме по трем измерениям: ширине, высоте и глубине (где глубина относится к третьему измерению объема, например количеству каналов в изображении или количеству каналов). фильтров в слое).

Чтобы сделать этот пример более конкретным, снова рассмотрим набор данных CIFAR-10: в ходе обучения будет иметь размеры $32 \times 32 \times 3$ (ширина, высота и глубина соответственно). Нейроны последующих слоев будут связываться только с небольшой областью слоя перед ним (а не с полностью связанный структурой стандартной нейронной сети) — мы называем это локальной связностью, которая позволяет нам со временем огромное количество параметров в нашей сети.. Наконец, в ходе обучения будет объем $1 \times 1 \times N$, который представляет изображение, преобразованное в единственный вектор оценок класса. В случае CIFAR-10 для десяти классов $N = 10$, что дает объем $1 \times 1 \times 10$.

11.2.1 Типы слоев

Существует много типов слоев, используемых для построения сверточных нейронных сетей, но вы скорее всего, столкнетесь с следующими:

- сверточный (CONV)
- активационный (ACT или RELU, где мы используем тоже самое, что и фактическая функция активации)
- объединение (POOL)
- Полностью подключенный (FC)
- Пакетная нормализация(BN)
- Отсев (DO)

Наложение ряда этих слоев определенным образом дает CNN. Мы часто используем простой текст диаграммы для описания CNN: INPUT => CONV => RELU => FC => SOFTMAX

Здесь мы определяем простую CNN, которая принимает в ходе обучения, применяет слой свертки, затем слой активации, затем полностью связанный слой и, наконец, классификатор softmax для получения в выходе вероятностей классификации. Уровень активации SOFTMAX часто опускается из сетевой диаграммы, поскольку предполагается, что он непосредственно следует за окончательным FC.

Из этих типов слоев CONV и FC (и в меньшей степени BN) являются единственными слоями, которые содержат параметры, которые изучаются процессом обучения. Уровни активации и отсева сами по себе не считаются настоящими «слоями», но часто являются сетевые диаграммы для ясности архитектуры. Уровни объединения (POOL), не менее важные, чем CONV и FC, также являются сетевые диаграммы, поскольку они имеют существенное влияние на пространственные размеры изображения при его перемещении по CNN.

CONV, POOL, RELU и FC являются наиболее важными при определении реальной сетевой архитектуры. Это означает, что другие уровни не являются критическими, но они отходят на второй план по сравнению с этим критическим набором из четырех, поскольку они определяют саму фактическую архитектуру.

 Практически предполагается, что сама функция активации является частью архитектуры. При определении архитектуры CNN мы часто опускаем слой активации из таблицы/диаграммы для экономии места; однако неявно предполагается, что уровни активации являются частью архитектуры.

В оставшейся части этого раздела мы подробно рассмотрим каждый из этих типов слоев и обсудим параметры свертывания с каждым слоем (и способами установки). Позже в этой главе я более подробно расскажу, как правильно сложить эти слои, чтобы построить собственную архитектуру CNN.

11.2.2 Сверточные слои Сверточный

слой является основным строительным блоком сверточной нейронной сети. Параметры слоя CONV состоят из набора K обучаемых фильтров (т. е. «ядер»), где каждый фильтр имеет ширину и высоту и почти всегда имеет квадратную форму. Эти фильтры малы (по своим пространственным размерам), но распространяются в свою глубину объема.

Для вх однък данык CNN глубина — это количество каналов в изображении (т. е. глубина равна трем при работе с RGB-изображениями, по одному на каждый канал). Для более глубоких томов в сети глубина будет равна количеству фильтров, примененных на предыдущем уровне.

Чтобы сделать эту концепцию более ясной, давайте рассмотрим прямой прох од CNN, где мы сворачиваем каждый из К фильтров по ширине и высоте вх одного объема, как мы это делали в разделе 11.1.5 выше. Процесс горя мы можем думать о каждом из наших К ядер, скользящем по вх одной области, вънисля щем поэлементное умножение, суммирующем, а затем сох ранго цем вък одное значение в двумерной карте активации, такой как на рис. 11.6.

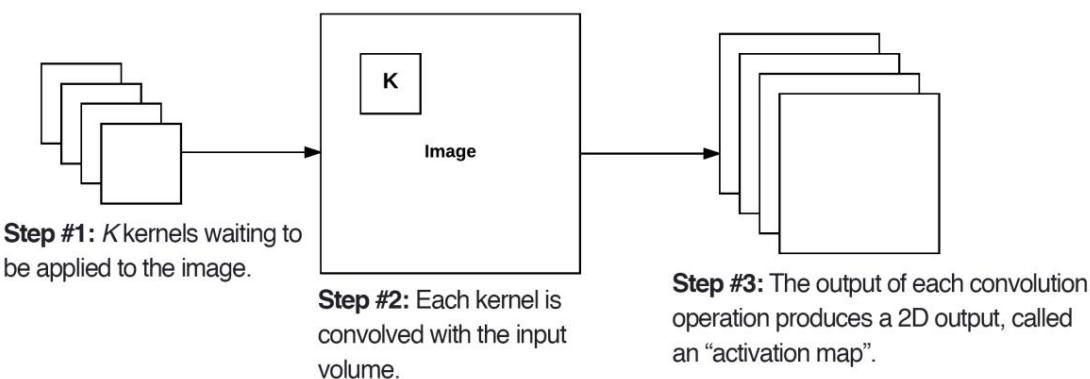


Рисунок 11.6: Слева: на каждом сверточном слое в CNN к вх одному объему применяется К ядер. Посредине: каждое из К ядер свё рнутос вх одnym объемом. Справа: Каждое ядро создает двухмерный вък од, называемый картой активации.

После применения всех К фильтров к вх одному объему теперь у нас есть К двумерных карт активации. Затем мы складываем наши К карт активации поизмерению глубины нашего массива, чтобы сформировать окончательный вък одной объем (рис. 11.7).

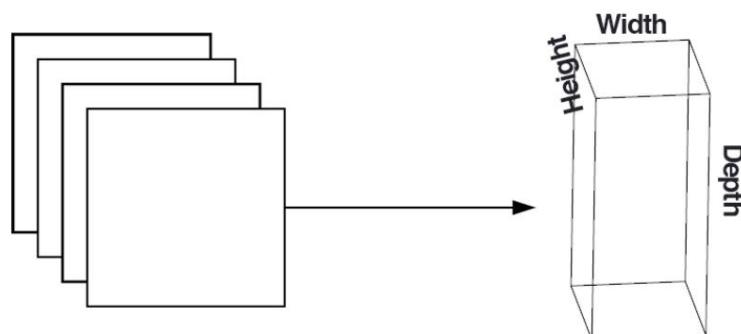


Рисунок 11.7: После получения К карт активации они складываются вместе, чтобы сформировать вх одной объем для следующего слоя сети.

Каждая запись в вък одном объеме, таким образом, является вък одн нейрона, который «смотрит» только на небольшую область вх од. Таким образом, сеть «изучает» фильтры которые активируются когда они видят определенный тип объекта в заданном пространственном местоположении в вх одном объеме. На нижних уровнях сети фильтры могут активироваться когда они видят краевые или угловые области.

Затем в более глубоких слоях сети фильтры могут активироваться при наличии признаков въсокого уровня, таких как части лица, лапа собаки, капот автомобиля и т. д. Эта концепция активации

всех одит к аналогии с нашей нейронной сетью в главе 10 — эти нейроны «воздушаются» и «активируются», когда видят определенный паттерн в одном изображении.

Концепция свертки небольшого фильтра с большим(r) в одном объеме имеет особое значение в сверточных нейронных сетях, в частности, для локальной связи и рецептивного поля нейрона. При работе с изображениями часто нецелесообразно соединять нейроны в текущем объеме со всеми нейронами предыдущего объема — просто слишком много связей и слишком много весов, что делает невозможным обучение глубоких сетей на изображениях с большими пространственными размерами. Вместо этого при использовании CNN мы выбираем соединение каждого нейрона только с локальной областью в одном объеме — мы называем размер этой локальной области рецептивным полем (или просто переменной F) в одного объема. нейрон

Чтобы прояснить этот момент, давайте вернемся к нашему набору данных CIFAR-10, где в одном объеме как в одной размер равен $32 \times 32 \times 3$. Таким образом, каждое изображение имеет ширину 32 пикселя в высоту 32 пикселя и глубину 3 (по одному для каждого канала RGB). Если наше рецептивное поле имеет размер 3×3 , то каждый нейрон в слое CONV будет соединяться с локальной областью изображения 3×3 , всего $3 \times 3 \times 3 = 27$ весов (помните, что глубина фильтров равна r , потому что они проходят через всю глубину в одном изображении в данном случае через три канала).

Теперь давайте предположим, что пространственные размеры нашего в одном объеме были уменьшены до меньшего размера, но наша глубина теперь больше из-за использования большего количества фильтров глубже в сети, так что размер тома теперь составляет $16 \times 16 \times 94$. Отметьте же, если предположить, что рецептивное поле имеет размер 3×3 , то каждый нейрон в слое CONV будет иметь в общей сложности $3 \times 3 \times 94 = 846$ соединений с в одном объеме. Процесс говоря, рецептивное поле F равно размеру фильтра, что дает ядро $F \times F$, свернутое в в одном объеме.

На данный момент мы обяснили связность нейронов в в одном объеме, но не расположение или размер в одном объеме. Есть три параметра, которые управляют размером в одном объеме: глубина, шаг и размер заполнения нулями, каждый из которых мы рассмотрим ниже.

Глубина

Глубина в одном объеме управляет количеством нейронов (то есть фильтров) в слое CONV, которые соединяются локальной областью в одном объеме. Каждый фильтр создает карту активации, которая «активируется» при наличии ориентированных краев, пятнистости цвета.

Для данного слоя CONV глубина карты активации будет равна K или просто количеству фильтров, которые мы изучаем в текущем слое. Набор фильтров, которые «смотрят» на одно и тоже (x, y) местоположение в одном объеме, называется стобцом глубины.

Страйд

Рассмотрим рисунок 11.1 ранее в этой главе, где мы описали операцию свертки как «скользящее» маленькой матрицы по большой матрице, останавливаясь на каждой координате, вычисляя поэлементное умножение и суммирование, а затем со временем результат. Это описание похоже на скользящее окно (<http://pyimg.co/0yiz0>), которое перемещается слева направо и сверху вниз по изображению.

В контексте Раздела 11.1.5 свертке выше мы сделали шаг только в один пиксель на каждую вершину. В контексте CNN можно применить тот же принцип — для каждого шага мы создаем новый столбец глубины в окрестности локальной области изображения, где мы сворачиваем каждый из K фильтров с областью и со временем результат в трехмерном объеме. При создании наших слоев CONV мы обычно используем размер шага S , равный $S = 1$ или $S = 2$.

Меньшие шаги приведут к перекрытию рецептивных полей и увеличению в одном объеме. И наоборот, большие шаги приведут к меньшему перекрытию рецептивных полей и меньшему объему продукции. Чтобы сделать концепцию сверточных шагов более конкретными, рассмотрим таблицу 11.1, где у нас есть в одном изображении 5×5 (слева) в месте с ядром Лагласа 3×3 (справа).

Используя $S = 1$, наше ядро скользит слева направо и сверху вниз, по одному пикселю за раз, создавая следующий результат (рис. 11.2, слева). Однако, если бы мы применили ту же операцию, только

95	242	186	152	39		
39	14	220	153	180	5	247
212	54	56	35	74	93	3160
						74

0	1	0		
1	-4	1		
0	1	0		

Таблица 11.1: Наше x одно изображение 5×5 (слева), которое мы собираемся свернуть с ядром Лагласа (справа).

692	-315	-6		
-680	-194	305		
153	-59	-86		

692	-6		
153	-86		

Таблица 11.2: Слева: Результат свертки с шагом 1×1 . Справа: результат свертки с шагом 2×2 . Обратите внимание, как больший шаг может уменьшить пространственные размеры в однотоманых.

на этот раз с шагом $S = 2$ мы пропускаем два пикселя за раз (два пикселя по оси x и два пикселя по оси y), создавая меньший в x одной объем (справа).

Таким образом, мы можем видеть, как слои свертки можно использовать для уменьшения пространственных размеров в однотоманых объемах, просто изменяя шаг ядра. Как мы видим позже в этом разделе, сверточные слои и объединяющие слои являются основными методами уменьшения пространственного размера в однотоманых. Рядом объединения слоев также предоставит более наглядный пример того, как изменение размера шага повлияет на x в однотоманом.

Заполнение

нулями Как мы знаем из Раздела 11.1.5, нам нужно «дополнить» границы изображения, чтобы сохранить исходный размер изображения при применении свертки — тоже самое верно для фильтров внутри CNN. Используя заполнение нулями, мы можем «дополнить» наш x в обе стороны границ, чтобы размер x в однотоманом соответствовал размеру в однотоманом. Количество применения этого правила контролируется параметром P .

Этот метод особенно важен, когда мы начинаем рассматривать глубокие архитектуры CNN, которые применяют несколько фильтров CONV друг над другом. Чтобы изучить заполнение нулями, снова обратитесь к таблице 11.1, где мы применили ядро Лагласа 3×3 к x одному изображению 5×5 с шагом $S = 1$.

В таблице 11.3 (слева) видно, что x в однотоманом (3×3) , чем x в однотоманом (5×5) из-за характера операции свертки. Если мы местом этого установим $P = 1$, мы можем дополнить наш x в однотоманом нулями (посредине), чтобы создать объем 7×7 , а затем применить операцию свертки, что приведет к размеру x в однотоманом, который соответствует исходному размеру в однотоманом 5×5 . (Правильно).

Без заполнения нулями пространственные размеры в однотоманом объеме будут уменьшаться слишком быстро, и мы не сможем обучать глубокие сети (поскольку в однотоманом объеме будут слишком малы, чтобы из них можно было извлечь какие-либо полезные закономерности).

Сложив все эти параметры вместе, мы можем вычислить размер x в однотоманом как функцию размера в однотоманом объеме (W , предполагая что x в однотоманом изображения квадратное, что почти всегда так), размера рецептивного поля F , шага S , и количества заполнения нулями P . Чтобы построить допустимый слой CONV, нам нужно убедиться, что следующее уравнение является целым числом:

$$((W-F+2P)/S)+1 \quad (11.6)$$

Если это не целое число, то шаги заданы неправильно, и нейроны могут быть замещены таким образом, чтобы они симметрично располагались в x однотоманом.

	0 0	0	0	0	0	0
692 -315 -6	0 95 242	186 152 39 0				
-680 -194 305	0 39 14 220	153 180 0				
153 -59 -86	0 5 247 212	54 46 0				
	0 46 77 133	110 74 0				
	0 156 35 74	93 116 0				
	0 0 0	0	0	0	0	
	-99	-673 -130 -230	176			
	-42 692 -315 -6	312		-482		
	-680 -194 305	124				
	54 153 -59 -86	-24				
	-543 167 -35 -72	-297				

Таблица 11.3: Слева: результат применения свертки 3×3 к вх оду 5×5 (т. е. пространственная размеры уменьшаются). Справа: применение нуляк исх одному вх оду с $P = 1$ увеличивает пространственные размеры до 7×7 . Внизу: после применения свертки 3×3 к дополненному вх оду наш время вх однотома соответствует исх одному размеру вх однотома 5×5 , поэтому нам помогает заполнение нулями . сох ранить пространственные размеры

В качестве примера рассмотрим первый уровень архитектуры AlexNet, победивший в 2012 г. Проблема классификации ImageNet, и она в значительной степени ответственна за нынешний бум глубокого обучения применяется к классификации изображений. В своей статье Крижевский и др. [94] задокументировали их CNN архитектуру согласно рисунку 11.8.

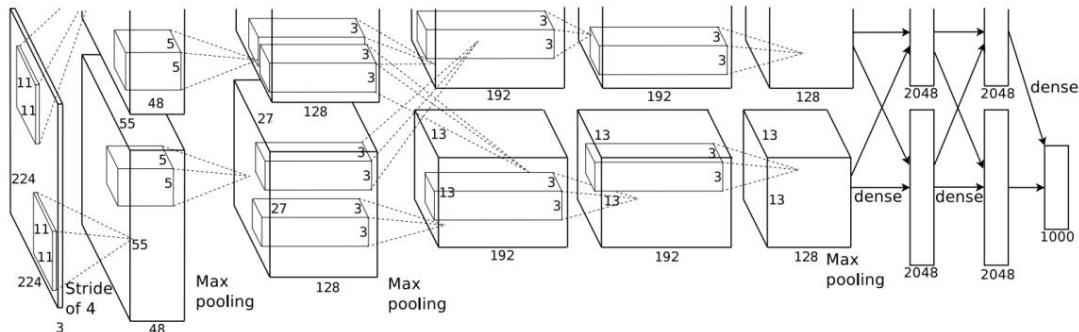


Рисунок 11.8: Первоначальная архитектура AlexNet, предоставленная Крижевским и др. [94]. Уведомление как вх одное изображение задокументировано как $224 \times 224 \times 3$, хотя это не возможно из-за Уравнение 11.6. Стоит также отметить, что мы не уверены почему в верхней половине этой цифры обрезана. из оригинальной публикации.

Обратите внимание, как первый слой утверждает, что размер вх одного изображения составляет 224×224 пикселя. Однако это не может быть правильным, если мы применим наше уравнение выше, используя фильтры 11×11 , шаг четыре и без прокладки:

$$((224 - 11 + 2(0))/4) + 1 = 54,25 \quad (11.7)$$

Что, конечно, не целое число.

Для начинаяющих читателей, которые только начинают изучать глубокое обучение и CNN, эта небольшая ошибка в такой оригинальной статье ввела бесчисленное количество ошибок, запутанных и разочарованных. Неизвестно, почему произошла эта ошибка, но вполне вероятно, что Крикевич и др. использовались в одни изображения 227x227, так как:

$$((227 - 11+2(0))/4) + 1 = 55 \quad (11.8)$$

Ошибки, подобные этим, встречаются часто, чем выдумаете, поэтому при реализации CNN из публикаций обязательно проверяйте параметры самостоятельно, а не просто предполагайте, что перечисленные параметры верны. Из-за огромного количества параметров в CNN довольно легко сделать ошибку при документировании архитектуры (я сам делал это много раз).

Подводя итог, слой CONV в той же элегантной манере, что и Karpachy [121]:

- Принимает вх одной объем размера Winput × Hinput × Dinput (размерах одни данные обычно квадратные,

поэтому часто можно увидеть Winput = Hinput).

- Требуется четыре параметра:

1. Количество фильтров K (конголирующих глубину вх одного объема).
2. Размер рецептивного поля F (размер ядер, используемых для свертки, и почти всегда квадратный, что дает ядро $F \times F$).
3. Страйд S .
4. Количество заполнения нулями P .

- Вх од слоя CONV тогда равен $W_{out} = (H_{in} + 2P)/S + 1$, где:

$$\begin{aligned} \text{- Ничего себе} &= ((\text{Вх од } F + 2P)/S) + 1 \\ \text{- Хватило} &= ((\text{Вх од } F + 2P)/S) + 1 \\ &= K \\ \text{- Даут помочь} \end{aligned}$$

Мы рассмотрим общие настройки этих параметров в разделе 11.3.1 ниже.

11.2.3 Уровни активации

После каждого слоя CONV в CNN мы применяем нелинейную функцию активации, такую как ReLU, ELU или любые другие варианты Leaky ReLU, упомянутые в главе 10. Мы обычно обозначаем уровни активации как RELU на сетевых диаграммах, поскольку активация ReLU является наиболее частой. Обычно используется мы также можем просто указать ACT — в любом случае мыясюдаем понять, что функция активации применяется внутри сетевой архитектуры.

Уровни активации технически не являются «слоями» (из-за того, что внутри уровня активации не изучаются никакие параметры веса) и иногда опускаются на диаграммах сетевой архитектуры поскольку предполагается, что активация следует сразу за сверткой.

В этом случае авторы публикаций будут упоминать, какую функцию активации они используют после каждого слоя CONV где-то в своей статье. В качестве примера рассмотрим следующую архитектуру сети: INPUT => CONV => RELU => FC.

Чтобы сделать эту диаграмму более лаконичной, мы могли бы просто удалить компонент RELU, так как предполагается, что активация всегда следует за сверткой: INPUT => CONV => FC. Лично мне это не нравится и я решил явно включить уровень активации в сетевую диаграмму, чтобы было ясно, когда и какую функцию активации я применяю в сети.

Уровень активации принимает вх одной объем размера Winput × Hinput × Dinput, а затем применяет заданную функцию активации (рис. 11.9). Поскольку функция активации применяется по элементу, вх одни данные слоя активации всегда совпадают с размерностью вх одни данные: Winput = Wout put, Hinput = Hout put, Dinput = Dout put.

11.2.4 Объединение слоев

Существует два метода уменьшения размера вх одного объема — слои CONV с шагом > 1 (чтобы учесть видели) и слои POOL. Обычно слои POOL вставляются между последовательными слоями.

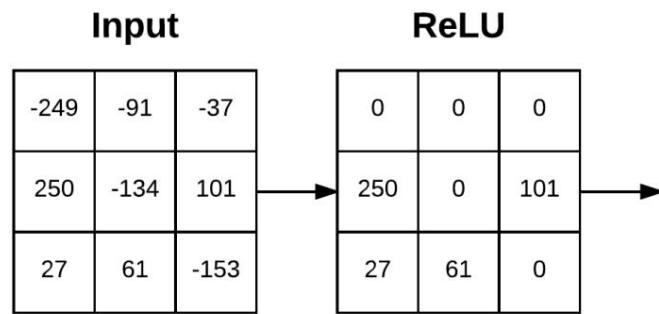


Рисунок 11.9: Пример вх однога объема, прох однога через активацию ReLU, $\max(0, x)$. Активации въполняются на месте, поэтому нет необходимости создавать отдельный вх однога объема, х отя таким образом легковизуализировать поток сети.

Уровни CONV в архитектурах CNN:

INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC

Основная функция слоя POOL заключается в постепенному уменьшении пространственного размера (т. е. ширины въготы) вх одного объема. Это позволяет нам уменьшить количество параметров и вычислений в сети — объединение также помогает нам контролировать переоснащение.

Слои POOL работают с каждым из срезов глубинных однок данных независимо, используя либо максимальную, либо среднюю функцию. Максимальный пул обычно выполняется середине архитектуры CNN для уменьшения пространственного размера, тогда как средний пул обычно используется качестве последнего уровня сети (например, GoogLeNet, SqueezeNet, ResNet), где мых отим полностью избежать использования слоев FC. Наиболее распространенным типом слоя POOL является максимальный пул, х отя эта тенденция меняется появлением более экзотических микроархитектур.

Обычно мы будем использовать размер пула 2×2 , х отя более глубокие CNN, которые используют более крупные вх однога изображения (> 200 пикселей), могут использовать размер пула 3×3 на ранней стадии сетевой архитектуры. Мы также обычно устанавливаем шаг либо $S = 1$, либо $S = 2$. На рис. 11.10 (в значительной степени вдохновленном Карпати и др. [121]) следует пример применения максимального объединения с размером пула 2×2 и шагом $S = 1$. Обратите внимание, что для каждого блока 2×2 мых раннем только самое большое значение, делаем один шаг (как скользящее окно) и применяем операцию снова — таким образом, получается размер вх одного тома 3×3 .

Мы можем дополнительно уменьшить размер нашего вх одного объема, увеличив шаг — здесь мы применяем $S = 2$ к тому же вх оду (рис. 11.10, внизу). Для каждого блока 2×2 вх однок мых раннем только наибольшее значение, затем делаем шаг в два пикселя и снова применяем операцию. Это объединение позволяет нам уменьшить ширину и въготу в два раза, эффективно отбрасывая 75% активаций из предыдущего слоя.

Таким образом, слои POOL принимают вх одного объема размера $W_{input} \times H_{input} \times D_{input}$. Затем они требуются два параметра:

- Размер рецептивного поля F (также называемый «размером пула»).
- Страйд C.

Применение операции POOL дает вх одного объема размером $W_{output} \times H_{output} \times D_{output}$, где:

$$\begin{aligned} W_{output} &= ((W_{input} - F)/S) + 1 \cdot W_{output} \\ D_{output} &= ((H_{input} - F)/S) + 1 \cdot H_{output} \\ &= B \times \text{од} \end{aligned}$$

На практике мысленно видеть два типа вариаций максимального объединения. Тип №1: $F = 3$, $S = 2$, который называется перекрывающимся объединением и обычно применяется к им.

возрастов вх однок объемов с большими пространственными размерами.

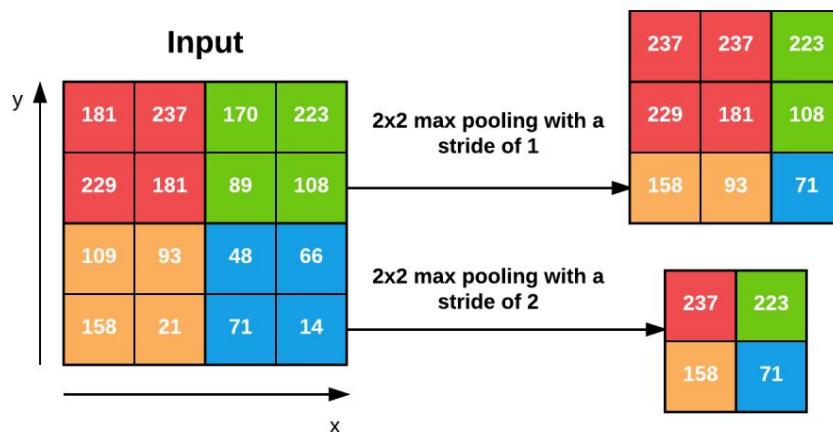


Рис. 11.10: Слева: объем ввода 4×4 . Справа: применение максимального объединения 2×2 с шагом $S = 1$. Внизу: применение максимального объединения 2×2 с $S = 2$ — это резко уменьшает пространственные размеры выходящего ввода.

- Тип № 2: $F = 2, S = 2$, который называется непрекрывающимся объединением. Это самый распространенный тип объединения и применяется к изображениям с меньшими пространственными размерами.

Для некоторых архитектур, которые принимают вх однъе изображения меньшего размера (в диапазоне 32-64 пикселя), вы также можете видеть $F = 2, S = 1$.

В БАССЕ ИН или CONV?

В своей статье 2014 года «Стремление к простоте: полностью сверточная сеть» Springenberg et al. [122] рекомендуют полностью отказаться от слоя POOL и просто оставить слои CONV с большим шагом, чтобы справиться с понижением дискретизации пространственных размеров объема. Их работа продемонстрирована, что этот подход однозначно работает с различными наборами данных, включая CIFAR-10 (маленькие изображения небольшое количество классов) и ImageNet (большие вх однъе изображения 1000 классов). Эта тенденция со временем распространяется в архитектуре ResNet [96], которая также использует слои CONV для понижения дискретизации.

Становится все более распространенным не использовать уровень POOL в середине сетевой архитектуры и использовать средний пул только в конце сети, если нужно избежать слоев FC. Возможно, в будущем сверточные нейронные сети не будут объединять слои, но пока важно, чтобы мы изучали их, узнавали, как они работают, и применяли их к нашим собственным архитектурам.

11.2.5 Полносвязные слои Нейронных слоев

FC полностью связанные с активациями на предыдущем уровне, что является стандартом для нейронных сетей с прямой связью, которые мы обсуждали в главе 10. Слои FC всегда располагаются в конце сетевой (т. е. мы не применяем слой CONV, затем слой FC, за которым следует еще один слой CONV).

Перед применением классификатора softmax обычно используют один или два слоя FC, как показано ниже. (упрощенная) архитектура демонстрирует:

INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => FC

Здесь мы применяем два полносвязных слоя перед нашим (подразумеваемым) классификатором softmax, который будет вычислять наши окончательные вх однъе вероятности для каждого класса.

11.2.6 Нормализация партии

В первые представление Иоффе и Седи в их статье 2015 года «Пакетная нормализация ускорение глубокого обучения» с помощью уменьшения внутреннего ковариатного сдвига» [123], уровня пакетной нормализации (или сокращенно BN), как следует из названия используются для нормализации активаций данных в одной объем перед передачей его на следующий уровень в сети.

Если мы считаем x_i нашей мини-партией активаций, то мы можем вычислить нормализованное значение \hat{x} через следующее уравнение:

$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sigma_\beta + \epsilon} \quad (11.9)$$

Во время обучения мы вычисляем μ_β и σ_β для каждой мини-партии β , где:

$$\mu_\beta = \frac{1}{M} \sum_{i=1}^M x_i \quad \sigma_\beta^2 = \frac{1}{M} \sum_{i=1}^M (x_i - \mu_\beta)^2 \quad (11.10)$$

Мы устанавливаем равным небольшому положительному значению, такому как 1e-7, чтобы не брать квадратный корень из нуля. Применение этого уравнения подразумевает, что активации, покидающие слой нормализации партии, будут иметь приблизительно нулевое среднее значение и единичную дисперсию (т. е. с нулевым центром).

Во время тестирования мы заменим мини-пакеты μ_β и σ_β скользящими средними значениями μ_β и σ_β , вычисленными в процессе обучения. Это гарантирует, что мы можем передавать изображения через нашу сеть и по-прежнему получать точные прогнозы без смещения из-за μ_β и σ_β из последней мини-партии, проходящей через сеть во время обучения.

Было показано, что пакетная нормализация является эффективной для сокращения количества эпох, необходимых для обучения нейронной сети. Пакетная нормализация также имеет дополнительное преимущество, помогая «стабилизировать» обучение, позволяя использовать большее разнообразие скоростей обучения и сильную сторону регуляризации. Использование пакетной нормализации, конечно, не избавляет от необходимости настраивать эти параметры, но облегчает вашу жизнь, сделав скорость обучения и регуляризацию менее изменчивыми и более простыми в настройке. Вы также заметите меньшие окончательные потери и более стабильную кривую потерь при использовании пакетной нормализации в своих сетях.

Самый большой недостаток пакетной нормализации заключается в том, что она может фактически замедлить время обучения одиночной для обучения нейронной сети (даже если вам потребуется меньше эпох для получения разумной точности) в 2-3 раза из-за вычисления статистики для каждой партии и нормализации.

Тем не менее, я рекомендую использовать пакетную нормализацию почти в каждой ситуации, поскольку она имеет существенное значение. Как мы увидим позже в этой книге, применение пакетной нормализации к нашим сетевым архитектурам может помочь нам предотвратить переоснащение и позволяет получить значительно более высокую точность классификации за меньшее количество эпох по сравнению с той же сетевой архитектурой без пакетной нормализации.

Итак, куда идут слои пакетной нормализации?

Вы наверное, заметили, что в моем обсуждении пакетной нормализации я не упомянул, где именно в сетевой архитектуре мы размещаем слой пакетной нормализации. Согласно оригинальной статье Иоффе и Седи [123], они поместили свою пакетную нормализацию (BN) перед активацией:

«Мы добавляем преобразование BN непосредственно перед нелинейностью, нормализуя $wu+b$ ».

Используя эту схему, сетевая архитектура, использующая пакетную нормализацию, будет выглядеть так:

INPUT => CONV => BN => RELU ...

Однако такое представление о пакетной нормализации не имеет смысла с статистической точки зрения. В этом контексте слой BN нормализует распределение признаков, включаяющих из слоя CONV. Некоторые из этих функций могут быть отрицательными, в которых они будут зафиксированы (т.е. обнулены) нелинейной функцией активации, такой как ReLU.

Если мы нормализуем перед активацией, мы по сути, включаем отрицательные значения в нормализацию. Наши функции с нулевым центром затем проходят через ReLU, где мы убиваем любые активации меньше нуля (включая функции, которые могли не быть отрицательными до нормализации) — такой порядок слоев полностью противоречит цели применения пакетной нормализации в первую очередь.

Вместо этого, если мы поместим пакетную нормализацию после ReLU, мы нормализуем функции с положительными значениями без статистического смещения с функциями, которые в противном случае не попали бы на следующий слой CONV. Фактически, Франсуа Шалле, создатель и сопровождающий Keras, подтверждает этот момент, заявляя, что BN должен появиться после активации:

«Я могу гарантировать, что недавний код, написанный Кристианом [Сегеди, из статьи BN], применяет relu перед BN. Тем не менее, иногда это все еще является предметом споров». [124]

Непонятно, почему Иоффе и Сегеди в своей статье предложили разместить слой BN перед активацией, но дальнейшие эксперименты [125], а также отдельные свидетельства других исследователей глубокого обучения [126] подтверждают, что размещение слоя пакетной нормализации после нелинейной активации дает более высокую точность и меньшие потери практических в восточных ситуациях.

Размещение BN после активации в сетевой архитектуре будет выглядеть так:

INPUT => CONV => RELU => BN ...

Я могу подтвердить, что почти во всех экспериментах, которые я проводил с CNN, размещение BN после RELU дает немногоболее высокую точность и меньшие потери. Тем не менее, обратите внимание на слово «почти» — было очень небольшое количество ситуаций, когда размещение BN до активации работало лучше, что означает, что вы должны умолчанию размещать BN после активации, но, возможно, захотите посвятить (максимум) один эксперимент по размещению BN перед активацией и записать результатов.

Проведя несколько таких экспериментов, вы быстро поймете, что BN после активации работает лучше, и в вашей сети есть более важные параметры, которые нужно настроить для получения более высокой точности классификации. Я обсуждаю это более подробно в разделе 11.3.2 далее в этой главе.

11.2.7 Отсева

Последний тип слоя, который мы собираемся обсудить, это отсев. Dropout на самом деле является формой регуляризации, целью которой является предотвращение переобучения за счет повышения точности тестирования возможно, за счет точности обучения. Для каждой мини- партии в нашем обучении наборе выпадающие слои с вероятностью p случайным образом отключаются от ключа, чтобы предыдущий слой следующему слою в сетевой архитектуре.

Рисунок 11.11 визуализирует эту концепцию, где мы случайным образом разъединяем с вероятностью $p = 0,5$ соединения между двумя уровнями FC для данного мини-пакета. Обратите внимание, что половина соединений для этого мини-пакета разорвана. После выполнения прямого и обратного прохода для мини-пакета мы повторно подключаем отброшенные соединения, а затем выбираем другой набор соединений для отображения.

Причина, по которой мы применяем отсев, состоит в том, чтобы уменьшить переоснащение путем явного изменения архитектуры сети в время обучения. Случайный сброс соединений гарантирует, что одинузел в сети не будет

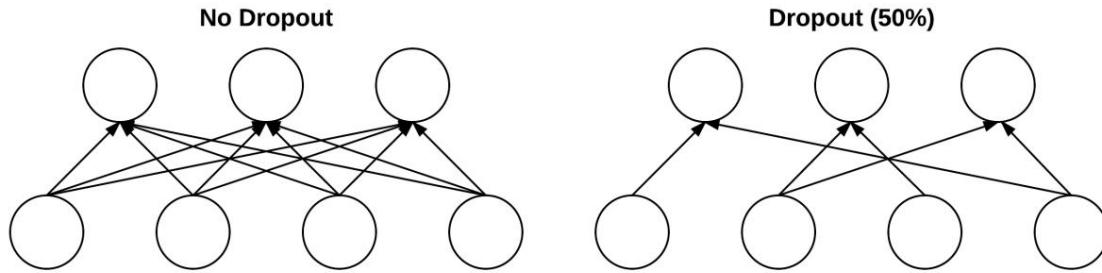


Рисунок 11.11: Слева: два слоя нейронной сети, которые полностью связаны без отсева. Справа: те же два слоя после удаления 50% соединений.

отвечает за «активацию» при представлении заданного шаблона. Вместо этого отсев гарантирует наличие нескольких избыточных узлов, которые будут активироваться при представлении аналогичных вх одноклассов — это, в свою очередь, помогает нашей модели обобщать.

Чаще всего между слоями FC в архитектуре размещают отсеваемые слои с $p = 0,5$.
где предполагается что последний слой FC является классификатором softmax:

... CONV => RELU => POOL => FC => DO => FC => DO => FC

Однако, как я обсуждаю в разделе 11.3.2, мы также можем применять отсев с меньшими вероятностями (т. е. $p = 0,10 \text{--} 0,25$) и на более ранних уровнях сети (обычно после операции понижения дискретизации либо посредством максимального объединения либо свертки) ..

11.3 Общие архитектуры и шаблоны обучения

Как мы видели в этой главе, сверточные нейронные сети состоят из четырех основных слоев: CONV, POOL, RELU и FC. Взятие этих слоев и объединение их вместе в определенном шаблоне дает архитектуру CNN.

Уровни CONV и FC (и BN) являются единственными уровнями сети, которые фактически изучают параметры — другие уровни просто отвечают за выполнение данной операции. Уровни активации (ACT), такие как RELU и dropout, технически не являются слоями, ночасто включаются в диаграммы архитектуры CNN, чтобы явно прояснить порядок операций — мы также применим такое же соглашение и в этом разделе.

11.3.1 Шаблоны слоев

На сегодняшний день наиболее распространенной формой архитектуры CNN является объединение нескольких слоев CONV и RELU с последующей операцией POOL. Мы повторим эту последовательность дотех пор, пока ширина и высота объема не станут небольшими, после чего мы применяем один или несколько слоев FC. Следовательно, мы можем ввести наиболее распространенную архитектуру CNN, используя следующий шаблон [121]:

INPUT => [[CONV => RELU]*N => POOL?]*M => [FC => RELU]*K => FC

Здесь оператор * подразумевает один или несколько, а оператор ? указывает на необязательную операцию.

Общие варианты выбора для каждой операции
включают:

- $0 \leq N \leq 3$
- $M \geq 0$

- $0 \leq K \leq 2$

Ниже мы можем увидеть несколько примеров архитектур CNN, которые следуют этому шаблону:

- INPUT => [CONV => RELU => POOL] * 2 => FC => RELU => FC

- INPUT => [CONV => RELU => CONV => RELU => POOL] * 3 => [FC => [FC => > RELU] * 2 => FC]

Вот пример очень мелкой CNN только с одним слоем CONV ($N = M = K = 0$), который мы рассмотрим в главе 12:

В В ОД => КОНВ => RELU => FC

Ниже приведен пример архитектуры CNN, подобной AlexNet [94], которая имеет несколько CONV => RELU => наборы слоев POOL, за которыми следуют слои FC:

INPUT => [CONV => RELU => POOL] * 2 => [CONV => RELU] * 3 => POOL =>
[FC => RELU => DO] * 2 => SOFTMAX

Для более глубоких сетевых архитектур, таких как VGGNet [95], мы будем размещать два (или более) уровня перед каждым уровнем POOL:

INPUT => [CONV => RELU] * 2 => POOL => [CONV => RELU] * 2 => POOL =>
[CONV => RELU] * 3 => POOL => [CONV => RELU] * 3 => POOL =>
[FC => RELU => DO] * 2 => SOFTMAX

Как правило, мы применяем более глубокую сетевую архитектуру, когда у нас (1) есть много полезных обучающих данных и (2) проблема классификации достаточно сложна. Наложение нескольких слоев CONV перед применением слоя POOL позволяет слоям CONV разрабатывать более сложные функции для выполнения операции деструктивного объединения.

Как мы узнаем из пакета ImageNet этой книги, существуют более «экзотические» сетевые архитектуры, которые отклоняются от этого шаблона и, в свою очередь, создают свои собственные шаблоны. Некоторые архитектуры полностью удаляют операцию POOL, полагаясь на уровень CONV для понижения дискретизации тома, а затем, в конце сети, применяется усреднение пула, а не слои FC для получения одних данных для классификаторов softmax.

Сетевые архитектуры, такие как GoogLeNet, ResNet и SqueezeNet [96, 97, 127], являются прекрасными примерами этого шаблона и демонстрируют, как удаление слоев FC приводит к меньшему количеству параметров и более быстрому обучению.

Эти типы сетевых архитектур также «накладываются» и объединяют фильтры по измерению канала: GoogLeNet применяет фильтры 1x1, 3x3 и 5x5, а затем объединяет их вместе по измерению канала для изучения многоуровневых функций. Отметь же, эти архитектуры считаются более «экзотичными» и считаются передовыми технологиями.

Если вас интересуют эти более продвинутые архитектуры CNN, обратитесь к ImageNet Bundle; в противном случае вам придется придерживаться базовых шаблонов наложения слоев, пока вы не изучите основы глубокого обучения.

11.3.2 Эмпирические правила

В этом разделе я рассмотрю общие практические правила построения собственных CNN. Для начала изображения представляются в одном слое, должны быть квадратными. Использование квадратных в одних данных позволяет нам использовать преимуществами библиотек оптимизации линейной алгебры. Общие размеры всех одного слоя включают 32x32,

64×64 , 96×96 , 224×224 , 227×227 и 229×229 (без учета количества каналов для удобства обозначений).

Во вторых, в ходе каждого слоя также должны кратно делиться на два после применения первой операции CONV. Вы можете сделать это, настроив размер фильтра и шаг. «Правило кратности на два» позволяет удобно и эффективно снижать выборку пространственных в одних данных в нашей сети с помощью операции POOL.

Как правило, ваши слои CONV должны использовать фильтры меньшего размера, такие как 3×3 и 5×5 . Крошечные фильтры 1×1 используются для изучения локальных функций, но только в более продвинутых сетевых архитектурах. Фильтры большего размера, такие как 7×7 и 11×11 , могут использоваться в качестве первого слоя CONV в сети (для уменьшения пространственного размера в одних данных при условии, что ваши изображения достаточно больше $> 200 \times 200$ пикселей); однако после этого начального слоя CONV размер фильтра должен резко уменьшиться иначе вы слишком быстро уменьшите пространственные размеры в объеме.

Вы также обычно используете шаг $S = 1$ для слоев CONV, по крайней мере, для меньших пространственных в одних объемах (сети, которые принимают большие в одних объемы, которые используют шаг $S \geq 2$ в первом слое CONV). Использование шага $S = 1$ позволяет нашим слоям CONV изучать фильтры в той же мере, как слой POOL отвечает за понижение дискретизации. Однако имейте в виду, что все сетевые архитектуры следуют этому шаблону — некоторые архитектуры вообще пропускают максимальный пул и полагаются на шаг CONV для уменьшения размера в объеме.

Лично я предпочитаю применять заполнение нулями к моим слоям CONV, чтобы гарантировать, что размер вектора одного измерения соответствует размеру вектора другого измерения — единственное исключение из этого правила — если я хочу намеренно уменьшить пространственные размеры с помощью свертки. Применение заполнения нулями при наложении нескольких слоев CONV друг на друга также продемонстрировано в практике повышение точности классификации. Как мы видим далее в этой книге, такие библиотеки, как Keras, могут автоматически вычислять для вас заполнение нулями, что еще больше упрощает построение архитектуры CNN.

Вторая общая рекомендация состоит в том, чтобы использовать слои POOL (а не слои CONV), чтобы уменьшить пространственные размеры в одних данных, по крайней мере, дотех пор, пока вы не станете более опытным в построении собственных архитектур CNN. Как только вы достигнете этой точки, вы должны начать экспериментировать с использованием слоев CONV, чтобы уменьшить пространственный размер в одних данных, и попытаться удалить максимальное количество слоев объединения из вашей архитектуры.

Чаше всего вы видите, что максимальное объединение применяется к размеру рецептивного поля 2×2 и шагу $S = 2$. Вы также можете увидеть рецептивное поле 3×3 на ранней стадии сетевой архитектуры, чтобы помочь уменьшить размер изображения. Крайне редко можно увидеть рецептивное поле размером более трех, поскольку эти операции очень разрушительны для одних данных.

Пакетная нормализация — дорогая операция, которая может удвоить или утроить время, необходимое для обучения CNN; однако рекомендуется использовать BN почти во всех ситуациях. Хотя BN действительно замедляет время обучения, он также имеет тенденцию «стабилизировать» обучение, упрощая настройку других гиперпараметров (конечно, есть некоторые исключения — я подробно описывал некоторые из этих «архитектур исключений» внутри ImageNet Bundle).

Я также размещую пакетную нормализацию после активации, что стало обычным явлением в сообществе глубокого обучения, хотя это противоречит оригинальной статье Иоффе и Сегеди [123].

При вставке BN в приведенные выше общие архитектуры уровней они становятся:

- INPUT => CONV => RELU => BN => FC.
- INPUT => [CONV => RELU => BN => POOL] * 2 => FC => RELU => BN => FC • INPUT => [CONV => RELU => BN => CONV => RELU => BN => ПУЛ] * 3 => [FC => RELU => BN] * 2 => FC

Вы можете применять пакетную нормализацию перед классификатором softmax, поскольку на данный момент мы предполагаем, что наша сеть изучила свои отличительные особенности раньше в архитектуре.

Отсев (DO) обычно применяется между слоями FC с вероятностью отсева 50% — вам следует рассмотреть возможность применения отсева почти в каждой архитектуре, которую вы создаете. Хотя это не всегда выполняется, мы также нравится включать слои отсева (с очень небольшой вероятностью, 10-25%) между слоями POOL и CONV. Из-за локальной связности слоев CONV отсев здесь менее эффективен.

ноя частонах одил это полезным в борьбе с переоснащением.

Помня об этих эмпирических правилах, вы сможете уменьшить головную боль при построении архитектур CNN, поскольку ваши слои CONV со временем имеют размеры вдвое меньше, как слои POOL помогут уменьшить пространственные размеры объемов, чтобы в конечном итоге приведет к слоям FC и окончательное вдвое уменьшение классификации.

После того, как вы освоите этот «традиционный» метод построения сверточных нейронных сетей, вы должны начать изучать возможность полного исключения операций максимального объединения и использования только слоев CONV для уменьшения пространственных размеров, чтобы в конечном итоге приведет к среднему объединению, а не к слою FC — эти типы более передовых методов архитектуры описаны в пакете ImageNet.

11.4. Инвариантны ли CNN к трансляции, вращению и масштабированию?

Обычный в опросе, который мне задают:

«Являются ли сверточные нейронные сети инвариантными к изменениям в переводе, вращению, и масштабированию? Поэтому они такие мощные классификаторы изображений?»

Чтобы ответить на этот вопрос, нам сначала нужно провести различие между отдельными фильтрами в сети и конечной обученной сетью. Отдельные фильтры CNN не инвариантны к изменениям в том, как повернуто изображение — мы демонстрируем это в главе 12 пакета ImageNet, где мы используем функции, извлеченные из CNN, чтобы определить, как ориентировано изображение.

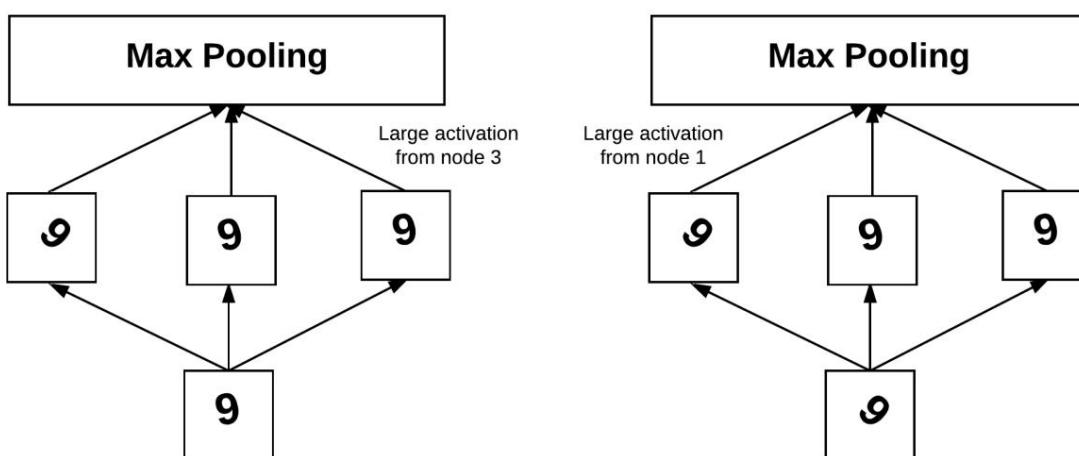


Рисунок 11.12: CNN в целом изучает фильтры, которые срабатывают, когда шаблон представлен в определенной ориентации. Слева цифра 9 повернута на 10°. Это вращение пока не на третий узел, который узел, как выглядит цифра 9 при вращении таким образом. Этот узел будет иметь более высокую активацию, чем два других узла — операция максимального объединения обнаружит это. Справа у нас есть второй пример, только на этот раз 9 повернута на 45°, в результате чего первая активация выше (рисунок в значительной степени взят из [10]).

Однако CNN в целом может обучаться фильтрам, которые срабатывают, когда шаблон представлен в определенной ориентации. Например, рассмотрим рисунок 11.12, адаптированный вдох новленный Deep Learning Гудфеллоу и др. [10].

Здесь мы видим цифру «9» (внизу), представленную CNN вместе с набором фильтров, которые CNN изучила (посередине). Поскольку внутри CNN есть фильтр, который «узнает», как выглядит цифра «9», повернутая на 10 градусов, он срабатывает и излучает сильную активацию. Эта большая активация фиксируется на этапе объединения в конечном итоге сообщается как окончательная классификация.

Тоже верно и для второго примера (рис. 11.12, слева). Здесь мы видим «9», повернутую на -45 градусов, и, поскольку в CNN есть фильтр, который узнал, как выглядит «9», когда она повернута на -45 градусов, нейрон активируется и срабатывает. Опять же, эти фильтры с самими собой не инвариантны вращению — просто CNN узнала, как выглядит «9» при небольших вращениях, которые существуют в обученном наборе.

Если ваши обученные данные не включают цифры, которые вращаются по всему спектру на 360 градусов, ваша CNN не является действительно инвариантной к вращению (опять же, этот момент продемонстрирован в главе 12 пакета ImageNet).

Тоже самое можно сказать и о масштабировании — сами фильтры не инвариантны масштабу, но восьмьма вероятно, что ваша CNN изучила набор фильтров, которые срабатывают, когда шаблоны существуют в различных масштабах. Мы также можем «помочь» нашей CNN быть инвариантными к масштабу, представив им наше примерное изображение в временных тестирования при различных масштабах и культурах, а затем средним результаты мест (см. точность).

трансляционная инвариантность; однако это то, в чем CNN преуспевает. Имейте в виду, что фильтр скользит слева направо и сверху вниз по вышесказанному и активируется, когда он сталкивается с определенной краевой областью, углом или цветным пятном. В временной операции объединения от большого ответа обнаруживаются, таким образом, «побеждает» в всех своих соседях, имея большую активацию. Следовательно, CNN можно рассматривать как «не заботящийся» от того, где именно срабатывает активация просто от того, что она действительно срабатывает, и, таким образом, мы естественным образом обрабатываем перевод внутри CNN.

11.5 Резюме

В этой главе мы знакомились с концепциями сверточных нейронных сетей (CNN). Мы начали с обсуждения того, что такое свертка и взаимная корреляция как эти термины заимствованы и используются в литературе по глубокому обучению.

Чтобы понять свертку на более близком уровне, мы реализовали ее вручную, используя Python и OpenCV. Однако радиционные операции обработки изображений требуют от нас ручного определения наших ядер и специфичных для данной задачи обработки изображений (например, сглаживание, обнаружение краев и т. д.). Используя глубокое обучение, мы можем вместе с этим изучать эти типы фильтров, которые затем накладываются друг на друга для автоматического обнаружения концепций на высоком уровне. Мы называем это наложением и изучением функций более высокого уровня на основе вхождения в один из данных более низкого уровня композиционностью сверточных нейронных сетей.

CNN строятся путем наложения последовательности слоев, где каждый слой отвечает за данную задачу. Слои CONV изучают набор из K сверточных фильтров, каждый из которых имеет размер $F \times F$ пикселей. Затем мы применяем слои активации поверх слоев CONV, чтобы получить нелинейное преобразование. Слои POOL помогают уменьшить пространственные размеры каждого объема при его проходе через сеть.

Как только в одном объеме становится достаточно маленьким, мы можем применить слой FC, которые являются нашими традиционными слоями скалярного произведения из главы 12, в конечном итоге вводящих в классификатор softmax для наших окончательных выходов одних прогнозов.

Слои пакетной нормализации используются для стандартизации входных данных для слоя CONV или активации путем вычисления среднего значения и стандартного отклонения дляミニ-пакета. Затем можно применить слой отсева для случайного отключения узлов от заданных входов, помогая уменьшить переоснащение.

Наконец, мы завершили главу обзором распространенных архитектур CNN, которые вы можете использовать для реализации своих собственных сетей. В нашей следующей главе мы реализуем нашу первую CNN в Keras, ShallowNet, на основе шаблонов слоев, которые мы упоминали выше. В следующих главах будут обсуждаться более глубокие сетевые архитектуры такие как основополагающая архитектура LeNet [19] и варианты архитектуры VGGNet [95].

12. Обучение вашего первого CNN

Теперь, когда мы рассмотрели основы сверточных нейронных сетей, мы готовы реализовать нашу первую CNN с использованием Python и Keras. Мы начнем главу с краткого обзора конфигураций Keras, о которых следует помнить при построении и обучении собственных CNN.

Затем мы реализуем ShallowNet, которая, как следует из названия, представляет собой очень неглубокую CNN только с одним слоем CONV. Однако не позволяйте простоте этой сети ввести вас в заблуждение — как показывают наши результаты ShallowNet способен обеспечить более высокую точность классификации как для CIFAR-10, так и для набора данных Animals, чем любой другой метод, рассмотренный нами до сих пор в этой книге..

12.1 Конфигурации Keras и преобразование изображений в массивы

Прежде чем мы можем реализовать ShallowNet, нам сначала нужно просмотреть файл конфигурации keras.json и то, как настройки внутри этого файла повлияют на то, как вы реализуете свои собственные CNN. Мы также реализуем второй preprocessор изображений под названием ImageToArrayPreprocessor, который принимает вх одное изображение, а затем преобразует его в массив NumPy, с которым может работать Keras.

12.1.1 Понимание файла конфигурации keras.json

Когда вы впервые импортируете библиотеку Keras в оболочку Python/выполняете скрипт Python, который импортирует Keras, за кулисами Keras создает файл keras.json в вашем домашнем каталоге. Вы можете найти этот файл конфигурации в `./keras/keras.json`.

Теперь откройте файл и посмотрите на его содержимое:

```
1 {  
2     «epsilon»: 1e-07,  
3     «floatx»: «float32»,  
4     «image_data_format»: «channels_last», «бэкенд»:  
5         «tensorflow»  
6 }
```

Вы заметите, что этот словарь в формате JSON имеет четыре ключа и четыре соответствующих значения. Значение эпсилон используется в различных местах библиотеки Keras для предотвращения деления на нули в ошибках. Значение по умолчанию 1e-07 подходит и его не следует изменять. Тогда у нас есть значение floatx, которое определяет точность с плавающей запятой — безопасно оставить это значение равным float32.

Последние две конфигурации, `image_data_format` и `backend`, чрезвычайно важны. По умолчанию библиотека Keras использует серверную часть численных вычислений TensorFlow. Мы также можем использовать бэкэнд Theano, просто заменив `tensorflow` на `theano`.

Вы должны помнить об этих бэкэндах при разработке собственных сетей глубокого обучения и когда вы развертываете их на других машинах. Keras отличается от других бэкэндов тем, что позволяет писать код глубокого обучения, который совместим с любым бэкэндом (и, конечно же, с более новыми бэкэндами). И поскольку большая часть вы обнаружите, что оба исследовательских бэкенда дают вам тот же результат. Если вы обнаружите, что ваши результаты не совпадают с вашим кодом возвращают ошибки, сначала проверьте свой сервер и убедитесь, что настройки соответствуют вашим ожиданиям.

Наконец, у нас есть `image_data_format`, который может принимать два значения `channels_last` или `channels_first`. Как мы знаем из предыдущих глав этой книги, изображения загружаются через OpenCV представлены в порядке (строки, столбцы, каналы), что называется `Keras channels_last`, так как каналы являются последним измерением массива.

В качестве альтернативы мы можем установить `image_data_format` как `channels_first`, где наши вх одни изображения представлены как (каналы строк, столбцы) — обратите внимание, что количество каналов указано первым измерением в массиве.

Почему две настройки? В соответствии с Theano пользователи, как правило, использовали каналы в порядке очереди. Однако, когда TensorFlow был выпущен в их руководствах и примерах, использовался последний порядок каналов. Это не соответствует введенной проблеме при использовании Keras в качестве кода, совместимого с Theano, потому что он может быть несовместим с TensorFlow в зависимости от того, как программист построил свою сеть. Таким образом, Keras представил специальную функцию `img_to_array`, которая принимает на вх од `image`, а затем правильно упорядочивает каналы на основе настройки `image_data_format`.

В общем, вы можете оставить настройку `image_data_format` как `channels_last`, и Keras позаботится о порядке измерений за вас независимо от бэкэнда; тем не менее, ях очу назвать это ситуацию в отдельном случае, если вы работаете с устаревшим кодом Keras и заметили, что используется другой порядок каналов изображения.

12.1.2 Препроцессор изображений в массивах

Как я уже упоминал выше, библиотека Keras предоставляет функцию `img_to_array`, которая принимает на вх од `image`, а затем правильно упорядочивает каналы на основе нашей настройки `image_data_format`. Мы собираемся поместить эту функцию в новый класс с именем `ImageToArrayPreprocessor`. Создание класса с помощью специальной функции предварительной обработки, точно так же, как мы делали это в главе 7 при создании `SimplePreprocessor` для изменения размера изображений, позволит нам создавать «цепочки» препроцессоров для эффективного подготовки изображений для обучения и тестирования.

Чтобы создать наш препроцессор преобразования изображений в массивы, создайте новый файл с именем `imagedatarraypreprocessor.py`. Внутри подмодуля предварительной обработки `pyimagesearch`:

```
|--- pyimagesearch
| |--- __init__.py
| |--- наборы данных
| | |--- __init__.py
| | |--- simpledatasetloader.py
| |--- предварительная обработка
| | |--- __init__.py
| | |--- imagedatarraypreprocessor.py
| | |--- simplepreprocessor.py
```

Оттуда откройте файл и вставьте следующий код:

```

1 # импортируем необходимые пакеты
2 из keras.preprocessing.image import img_to_array
3
4 class ImageToArrayPreprocessor: def
5     __init__(self, dataFormat=None): # сохраниТЬ
6         формат_данных_изображения
7         self.dataFormat = dataFormat
8
9     def preprocess(self, image): #
10        применить служебную функцию Keras, которая корректно
11        переупорядочивает # размеры изображения return img_to_array(image,
12        data_format=self.dataFormat)

```

Строка 2 импортирует функцию `img_to_array` из Keras.

Затем мы определяем конструктор для нашего класса `ImageToArrayPreprocessor` в строках 5-7. Конструктор принимает необязательный параметр с именем `dataFormat`. По умолчанию это значение равно `None`, что означает, что следует использовать настройку внутри `keras.json`. Мы также могли бы явно указать строки `channels_first` или `channels_last`, которые позволяют Keras выбирать, какой порядок измерения изображения использовать на основе файла конфигурации.

Наконец, у нас есть функция предварительной обработки в строках 9-12. Этот метод: 1. Принимает изображение в качестве входных данных.

2. Выводит `img_to_array` для изображения, упорядочивая каналы на основе нашего конфигурационного файла / значение формата данных .

3. Возвращает новый массив NumPy с правильным упорядочением каналами.

Преимущество определения класса для обработки этого типа предварительной обработки изображений, а не простого вызова `img_to_array` для каждого отдельного изображения заключается в том, что теперь мы можем связывать преобразователи изображений с нашим классом `ImageToArrayPreprocessor`.

Например, предположим, что мы хотим изменить размер всех входных изображений до фиксированного размера 32×32 пикселей. Для этого нам потребуется инициализировать наш `SimpleProcessor` из главы 7:

1 sp = простой преобразователь (32, 32)

После изменения размера изображения нам нужно применить правильный порядок каналов — это может быть выполнено с использованием нашего `ImageToArrayPreprocessor` выше:

2 iap = ImageToArrayPreprocessor()

Теперь предположим, что мы хотим загрузить набор данных изображений с диска и подготовить все изображения в наборе данных для обучения. Используя `SimpleDatasetLoader` из главы 7, наша задача становится очень простой:

3 sdl = SimpleDatasetLoader (преобразователи= [sp, iap]) 4 (данные, метки) = sdl.load (imagePaths, verbose = 500)

Обратите внимание, как наши преобразователи изображений связаны друг с другом и будут применяться в последовательном порядке. Для каждого изображения в нашем наборе данных мы сначала применим `SimplePreprocessor`, чтобы изменить его размер до

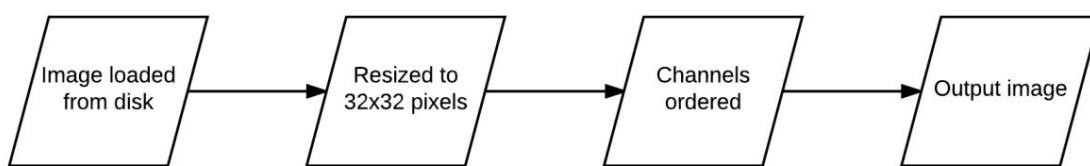


Рисунок 12.1: Пример конвейера предварительной обработки изображения который (1) загружает изображение с диска, (2) изменяет его размер до 32×32 пикселей, (3) упорядочивает размещение каналов и (4) выводит изображение.

32×32 пикселя После изменения размера изображения для обработки применяется преобразователь ImageToArrayPreprocessor. Упорядочивание каналов изображения Этот конвейер обработки изображений можно визуализировать на рисунке 12.1.

Объединение простых преобразователей вместе таким образом, когда каждый преобразователь отвечает с одной стороны небольшая работа, — это простой способ создать расширяемую библиотеку глубокого обучения, посвященную классификации изображений. Мы будем использовать эти преобразователи в следующем разделе, а также определим более продвинутые преобразователи в пакетах Practitioner Bundle и ImageNet Bundle.

12.2 Мелкая сеть

В этом разделе мы реализуем архитектуру ShallowNet. Как следует из названия Архитектура ShallowNet содержит всего несколько уровней — вся система архитектура может быть резюмирована как: INPUT => CONV => RELU => FC

Эта простая архитектура позволит нам приступить к реализации Convolutional. Нейронные сети с использованием библиотеки Keras. После реализации ShallowNet я применил его к Животные и наборы данных CIFAR-10. Как продемонстрируют наши результаты CNN способен значительно превосходить предыдущие методы классификации изображений, обсуждаемые в этой книге.

12.2.1 Реализация мелкой сети

Чтобы соединить наш пакет pyimagesearch в чистоте, давайте создадим внутри него новый подмодуль с именем conv, где будут жить все наши реализации CNN:

```

--- pyimagesearch
| |--- __init__.py
| |--- наборы данных
| |--- нн
| | |--- __init__.py
...
| |--- конв.
| | |--- __init__.py
| | |--- мелкая сеть.py
| --- предварительная обработка
  
```

Внутри подмодуля conv создайте новый файл с именем smallnet.py для реализации нашей ShallowNet. архитектуры. Откройте файл и вставьте следующий код:

```

1 # импортируем необходимые пакеты
2 из импорта keras.models Последовательный
3 из keras.layers.convolutional и импорт Conv2D
  
```

```
4 из keras.layers.core импорт Активация
5 из keras.layers.core import Flatten
6 из keras.layers.core импорт Плотный
7 из keras импортируем бэкенд как K
```

Строки 2-7 импортируют необязательные пакеты Python. Класс Conv2D — это реализация Keras сверточный слой обсуждался в разделе 11.1. Затем у нас есть класс Activation, который, как следует из названия, обрабатывает применение функции активации к входу. Классы Flatten занимают наш многомерный объем и «выврывают» его в одномерный массив перед подачей в один или несколько данных в Плотные (т.е. полно связанные) слои.

При реализации сетевых архитектур я предполагаю определять их внутри класса, чтобы соединить код организован — мы сделаем тоже самое здесь:

```
9 класс Мелкаясеть:
10     @staticmethod метод
11     def build(ширина, высота, глубина, классы):
12         # инициализируем модель вместе с вх одной формой, чтобы она была
13         # "каналыпоследние"
14         модель = Последовательный()
15         inputShape = (высота, ширина, глубина)
16
17         # если мы используем "сначала каналы", обновите форму ввода
18         если K.image_data_format() == "channels_first":
19             inputShape = (глубина, высота, ширина)
```

В строке 9 мы определяем класс ShallowNet, а затем определяем метод сборки в строке 11. Каждый CNN, которую мы реализуем в этой книге, будет иметь метод сборки — эта функция будет принимать числовые параметры, построить сетевую архитектуру, а затем вернуть ее в возвращающую функцию.

В этом случае наш метод сборки требует четырех параметров:

- ширина: ширина вх одних изображений, которые будут использоваться для обучения сети (т.е. количество столбцов матрицы).
- высота: высота наших вх одних изображений (т.е. количество строк в матрице).
- глубина: количество каналов в вх одних изображениях.
- классы: количество классов, которые наша сеть должна научиться предсказывать. Для живых объектов, классы=3, а для CIFAR-10 классы=10.

Затем мы инициализируем inputShape для сети в строке 15, предполагая порядок «каналыпоследними».

Строки 18 и 19 проверяют, установлен ли бэкенд Keras на «сначала каналы», и если да, то мы обновим inputShape. Обычной практикой является включение строк 15-19 почти для каждой CNN, которая выстроите, тем самым гарантируя, что наша сеть будет работать независимо от того, как пользователь заказывает каналы своего изображения.

Теперь, когда наш inputShape определен, мы можем начать строить архитектуру ShallowNet:

```
21     # определяем первый (и единственный) слой CONV => RELU
22     model.add(Conv2D(32, (3, 3), padding="тоже самое",
23                     input_shape=inputShape))
24     model.add(Активация("relu"))
```

В строке 22 мы определяем первый (и единственный) сверточный слой. Этот слой будет иметь 32 фильтра (K) каждый из которых имеет размер 3x3 (т.е. квадратные фильтры FxF). Мы будем применять те же отступы, чтобы обеспечить размер ввода операции свертки соответствует вводу (использование одного и того же заполнения является строго необязательным).

для этого примера, но это означает, что мы начали формировать сеть сейчас). После свертки мы применяем активацию ReLU в строке 24.

Закончим сборку ShallowNet:

```

26         # классификатор softmax
27         model.add(Flatten())
28         model.add(Dense(classes))
29         model.add(Activation("softmax"))
30
31     # вернуть построенную сетевую архитектуру
32     модель возврата

```

Чтобы применить наш полноценный слой, нам сначала нужно преобразовать многомерное представление в одномерный список. Операция `flatten` обрабатывается в строке 27. Затем создается плотный слой с использованием того же количества узлов, что и метки нашего вектора одного класса (строка 28). Стока 29 применяет функцию активации softmax, которая дает нам вероятности меток классов для каждого класса. Архитектура ShallowNet возвращается в вьюшую функцию в строке 32.

Теперь, когда ShallowNet определен, мы можем перейти к созданию реальных «скриптов драйверов», используемых для загрузки набора данных, его предварительной обработки и последующего обучения сети. Мы рассмотрим два примера использования ShallowNet — Animals и CIFAR-10.

12.2.2 ShallowNet на животных

Чтобы обучить ShallowNet на наборе данных Animals, нам нужно создать отдельный файл Python. Откройте свою любимую IDE, создайте новый файл с именем `smallnet_animals.py`, убедившись, что он находится на том же уровне каталога, что и наш модуль `pyimagesearch` (или вы добавили `pyimagesearch` в список путей, который вашингерпретатор / IDE Python будет проверять при запуске скрипта.).

Оттуда мы можем приступить к работе:

```

1 # импортируем необх одимые пакеты2 из
sklearn.preprocessing импортируем LabelBinarizer 3 из sklearn.model_selection
import train_test_split 4 из sklearn.metrics ----- pyimagesearch.nn.conv
импортировать ShallowNet 9 из keras.optimizers импортировать SGD 10 из imutils
импортировать пути 11 импортировать matplotlib.pyplot как plt 12 импортировать numpy как np 13
импортировать argparse

```

Строки 2-13 импортируют необх одимые пакеты Python. Большинство этих операций импорта вы видели в предыдущих примерах, но я хочу обратить ваше внимание на строки 5-7, где мы импортируем наши препроцессоры `ImageToArrayPreprocessor`, `SimplePreprocessor` и `SimpleDatasetLoader` — эти классы формируют фактический конвейер, используемый для обработки изображений перед их передачей через нашу сеть. Затем мы импортируем ShallowNet в строку 8 вместе с SGD в строку 9 — мы будем использовать стохастический градиентный спуск для обучения ShallowNet.

Затем нам нужно проанализировать аргументы командной строки и получить пути к нашим изображениям:

```

15 # построить анализатор аргументов и проанализировать аргументы
16 ap = argparse.ArgumentParser() 17 ap.add_argument("-d", "--dataset",
required=True,
18     help="путь к вх одному набору
данник ") 19 args = vars(ap.parse_args())
20
21 # получить список изображений, которые мы будем описывать
22 print("[INFO] загрузка изображений...") 23 imagePaths =
list(paths.list_images(args["dataset"]))

```

Наш скрипт требует здесь только одного ключа, --dataset, который представляет собой путь к каталогу, содержащему наш набор данных Animals. Затем в строке 23 берутся почти все 3000 изображений внутри Animals.

Помните, я говорил о создании конвейера для загрузки и обработки нашего набора данных? Давайте посмотрите, как это делается сейчас:

```

25 # инициализируем преобразователи
изображений 26 sp = SimplePreprocessor(32, 32) 27
iap = ImageToArrayPreprocessor()
28
29 # загрузить набор данных с диска, затем масштабировать необработанные
изображения 30 # диапазон [0, 1] 31 sdl =
SimpleDatasetLoader(preprocessors=[sp, iap]) 32 (data, labels) = sdl.load(imagePaths,
verbose=500) 33 данные = data.astype("с плавающей запятой") / 255.0

```

Строка 26 определяет SimpleProcessor, используемый для изменения размера всех изображений до 32×32 пикселей. Затем в строке 27 создается экземпляр ImageToArrayPreprocessor для обработки порядка каналов.

Мы объединяем эти преобразователи вместе в строке 31, где мы инициализируем SimpleDatasetLoader. Взгляните на параметр преобразователей конструктора — мы предоставляем список преобразователей, которые будут применяться в последовательном порядке. В первый, заданное в ходе изображение будет изменено до размера 32×32 пикселя. Затем каналы изображения с измененным размером будут упорядочены согласно нашим конфигурационным файлом keras.json. Страна 32 загружает изображения (применив преобразователи) и метки классов. Затем мы масштабируем изображения до диапазона [0, 1].

Теперь, когда данные и метки загружены, мы можем выполнить разделение обучения и тестирования, а также с однократным кодированием меток:

```

35 # разбить данные на обучающие и тестовые сплиты используя 75% из 36 # данных
для обучения и оставшиеся 25% для тестирования 37 (trainX, testX, trainY, testY) =
train_test_split(data, labels, test_size=0.25, random_state=42)
38
39
40 # преобразовать метки из целых чисел в векторы 41
trainY = LabelBinarizer().fit_transform(trainY) 42 testY =
LabelBinarizer().fit_transform(testY)

```

Здесь мы используем 75% наших данных для обучения и 25% для тестирования.

Следующим шагом является создание экземпляра ShallowNet с последующим обучением самой сети:

```

44 # инициализируем оптимизатор и модель
45 print("[INFO] компиляция модели...") 46 opt =
SGD(lr=0.005) 47 model = ShallowNet.build(width=32,
height=32, depth=3, классы=3) 48 model.compile(loss="categorical_crossentropy",
оптимизатор=opt,
49     метрики=["точность"])
50
51 # обучаем сеть
52 print("[INFO] обучаю сеть...")
53 H = model.fit(trainX, trainY, validation_data=(testX, testY),
batch_size=32, эпох=100, подсчет=1)

```

Мы инициализируем оптимизатор SGD в строке 46, используя скорость обучения 0,005 (мы будем, как настроить скорость обучения в следующей главе). Архитектура ShallowNet создается в строке 47, предоставляя ширину и высоту 32 пикселя вместе с глубиной 3 — это означает, что наши вх однъе изображения имеют размер 32 × 32 пикселя с тремя каналами. Поскольку набор данных Animals имеет три метки классов, мы устанавливаем class=3.

Затем модель компилируется в строках 48 и 49, где мы будем использовать кросс-энтропию в качестве нашей функции потерь и SGD в качестве нашего оптимизатора. Чтобы на самом деле обучить сеть, мы вызываем метод модели .fit в строках 53 и 54. Метод .fit требует от нас передачи данных обучения и тестирования. Мы также предоставляем данные тестирования, чтобы мы могли оценивать производительность ShallowNet после каждой эпохи. Сеть будет обучена для 100 эпох с использованием мини-пакетов размером 32 (это означает, что 32 изображения будут представлены сеть одновременно, и будет выполнен полный прямой и обратный проход для обновления параметров сети).

После обучения нашей сети мы можем оценить ее производительность:

```

56 # оценим сеть
57 print("[INFO] оценив сеть...") 58 предсказания =
model.predict(testX, batch_size=32) 59 print(classification_report(testY.argmax(axis=1),
60     прогнозы=argmax(ось=1),
61     target_names=["кошка", "собака", "панда"]))

```

Чтобы получить вх однъе прогнозы для наших данных тестирования, мы вызываем .predict модели. Хорошо отформатированный отчет о классификации отображается на нашем экране в строках 59-61.

Наш последний блок кода обрабатывает график точности и потерь с течением времени как для данных обучения так и для данных тестирования.

```

63 # график потерь и точности обучения
plt.style.use("ggplot") 65 plt.figure() 66
plt.plot(np.arange(0, 100), H.history["loss"], label=
"train_loss") 67 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss") 68
plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc") 69 plt.plot(np.arange(0,
100), H.history["val_acc"], label="val_acc") 70 plt.title("Потери при обучении и
точность") 71 plt.xlabel("Эпохи") 72 plt.ylabel("Потери/точность") 73 plt.legend() 74
plt.show()

```

Чтобы обучить ShallowNet на наборе данных Animals, просто выполните следующую команду:

```
$ python smallnet_animals.py --dataset ..datasets/animals
```

Обучение должно быть довольно быстрым, так как сеть очень неглубокая и наш набор данных изображений относительно мал. небольшой:

```
[INFO] загрузка изображений...
[INFO] обработано 500/3000
[INFO] обработано 1000/3000
[INFO] обработано 1500/3000
[INFO] обработано 2000/3000
[INFO] обработано 2500/3000
[INFO] обработано 3000/3000
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 2250 образцах, проверка на 750 образцах
Эпох 1/100
0s - убыток: 1,0290 - акк: 0,4560 - val_loss: 0,9602 - val_acc: 0,5160
Эпох 2/100
0s - убыток: 0,9289 - акк: 0,5431 - val_loss: 1,0345 - val_acc: 0,4933
...
Эпох 100/100
0s - убыток: 0,3442 - акк: 0,8707 - val_loss: 0,6890 - val_acc: 0,6947
[INFO] оценка сети...
      Точность      Вспоминать поддержку f1-score
    Кот        0,58        0,77        0,67        239
              0,75        0,40        0,52        249
    собака панда  0,79        0,90        0,84        262
    среднее / общее  0,71        0,69        0,68        750
```

Из-за небольшого количества обучающих данных эпохи были довольно быстрыми, занимая менее одной секунды как на нашем процессоре, так и на графическом процессоре.

Как видно из приведенного выше ввода, ShallowNet получил точность классификации 71% на наши данные тестирования значительное улучшение по сравнению с нашим предыдущим лучшим результатом в 59% с использованием простой прямой сводки нейронные сети. Используя более продвинутые обучающие сети, а также более мощную архитектуру, мы можем повысить точность классификации еще выше.

График потерь и точности со временем показан на рис. 12.2. На оси X у нас есть номер эпохи, а по оси Y у нас есть наши потери и точность. Рассматривая этот рисунок, мы видим что обучение немного изменчиво с большими всплесками потерь в эпохе 20 и эпохе 60 - этот результат вероятно, из-за того, что наша скорость обучения слишком высока, и мы можем решить эту проблему в главе 16.

Также обратите внимание, что потери при обучении и тестировании сильно расходятся после 30-й эпохи, что подразумевает что наша сеть слишком точно моделирует тренировочные данные и переоснащает их. Мы можем решить эту проблему путем получения большего количества данных или применения таких методов, как увеличение данных (рассматривается в Практике Пункта).

Примерно к эпохе 60 точность нашей тестирований достигает предела — мы не можем преодолеть 70% классификации. Точность, в свою очередь, как наша точность обучения продолжает расти и превышает 85%. Отметьте же, собирая больше обучающих данных, применение дополнений к данным и более щадящая настройка скорости обучения помогут нам улучшить наши результаты будущим.

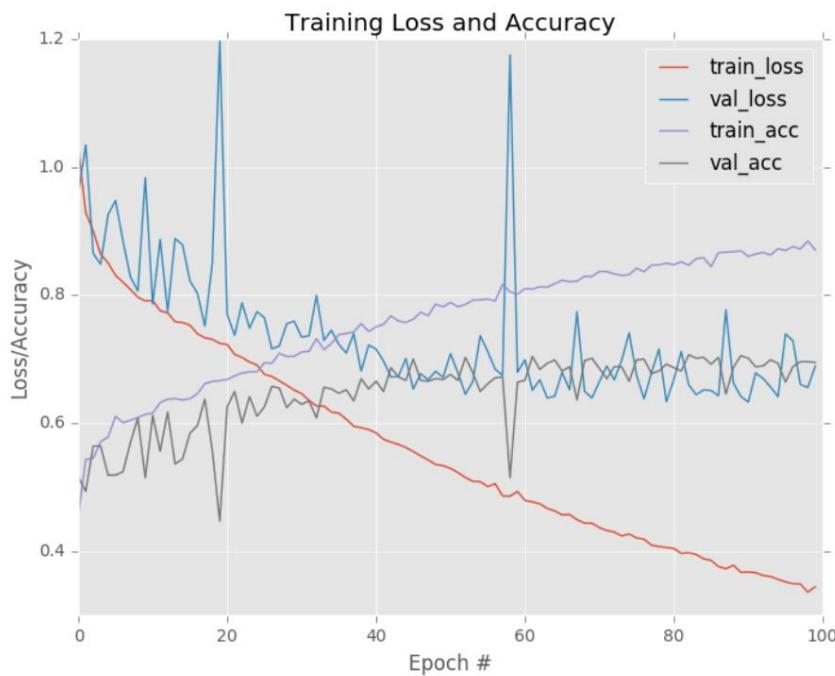


Рисунок 12.2: График наших потерь и точности в течение 100 эпох для архитектуры ShallowNet, обученной на наборе данных Animals.

Ключевым моментом здесь является то, что впервые просто сверточная нейронная сеть смогла получить точность классификации 71% в наборе данных животных, тогда как наш предыдущий лучший показатель составлял всего 59% — это улучшение более чем на 12%!

12.2.3 ShallowNet на CIFAR-10

Давайте также применим архитектуру ShallowNet к набору данных CIFAR-10, чтобы посмотреть, сможем ли мы улучшить наши результаты. Откройте новый файл, назовите его smallnet_cifar10.py и вставьте следующий код:

```
1 # импортируем необходимые пакеты
из sklearn.preprocessing импортируем LabelBinarizer 3 из
sklearn.metrics импортируем classification_report 4 из
pyimagesearch.nn.conv импортируем ShallowNet 5 из
keras.optimizers импортируем SGD 6 из keras.datasets
импортируем cifar10 7 импортируем matplotlib.pyplot as plt 8
импортировать numpy как пр
```

⁹
10 # загрузить данные обучения и тестирования, затем масштабировать их
в диапазоне [0, 1] 11 # [0, 1] 12 print("[INFO] загрузка данных CIFAR-10...") 13 ((trainX,
trainY), (testX, testY)) = cifar10.load_data() 14 trainX = trainX.astype("float") / 255.0 15
testX = testX.astype("float") / 255.0

¹⁶
17 # преобразовать метки из целых чисел в векторы 18
lb = LabelBinarizer()

```

19 trainY = lb.fit_transform(trainY) 20 testY =
lb.transform(testY)
21
22 # инициализировать имена меток для набора данных CIFAR-10
23 labelNames = ["самолет", "автомобиль", "птица", "кошка", "олень", "собака",
24     "лягушка", "лошадь", "корабль", "грузовик"]

```

Строки 2-8 импортируют необъекты Python. Затем мы загружаем набор данных CIFAR-10 (предварительно разделенный на наборы для обучения и тестирования), после чего масштабируем интенсивность пикселей изображения до диапазона [0,1]. Поскольку изображения CIFAR-10 предварительно обрабатываются в порядке каналов обрабатывается автоматически внутри cifar10.load_data, нам не нужно применять какие-либо из наших пользовательских классов предварительной обработки.

Затем наши метки сразу кодируются в векторы в строках 18-20. Мы также инициализируем метку имена для набора данных CIFAR-10 в строках 23 и 24.

Теперь, когда наши данные подготовлены, мы можем обучить ShallowNet:

```

26 # инициализируем оптимизатор и модель
27 print("[INFO] компиляция модели...") 28 opt =
SGD(lr=0.01) 29 model = ShallowNet.build(width=32,
height=32, depth=3, классы=10) 30 model.compile(loss="categorical_crossentropy",
оптимизатор=opt,
31     метрики=["точность"])
32
33 # обучаем сеть
34 print("[INFO] обучая сеть...")
35 H = model.fit(trainX, trainY, validation_data=(testX, testY),
36     batch_size=32, эпох=40, подробный=1)

```

Строка 28 инициализирует оптимизатор SGD со скоростью обучения 0,01. Затем ShallowNet строится на линии 29, используя ширину 32, высоту 32 и глубину 3 (поскольку изображения CIFAR-10 имеют три канала). Мы устанавливаем class=10, так как, как следует из названия в наборе данных CIFAR-10 десять классов. Модель компилируется в строках 30 и 31, а затем обучается в строках 35 и 36 в течение 40 эпох.

Оценка ShallowNet выполняется точно так же, как и в предыдущем примере с набором данных оживотных:

```

38 # оценить сеть
39 print("[INFO] оценка сети...") 40 прогнозы=
model.predict(testX, batch_size=32) 41
print(classification_report(testY.argmax(axis=1),
42     прогнозы.argmax(axis=1), target_names=labelNames))

```

Мы также будем отображать потери и точность со временем, чтобы понять, как работает наша сеть:

```

44 # график потери и точности обучения 45
plt.style.use("ggplot") 46 plt.figure() 47
plt.plot(np.arange(0, 40), H.history["loss"], label=
"train_loss") 48 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")

```



Рисунок 12.3: Потери и точность для ShallowNet, обученного на CIFAR-10. Наша сеть обеспечивает точность классификации 60%; однако это переоснащение. Дополнительную точность можно получить, применяя регуляризацию, которую мы рассмотрим позже в этой книге.

```
49 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc") 50 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc") 51 plt.title("Потери и точность обучения") 52
plt.xlabel("Эпох а #") 53 plt.ylabel("Потери/точность") 54 plt.legend() 55 plt. шоу()
```

Чтобы обучить ShallowNet на CIFAR-10, просто выполните следующую команду:

```
$ python smallnet_cifar10.py [INFO]
загружает данные CIFAR-10...
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 50000 выборках, проверка на 10000 выборках
Эпох а 1/40 5 с - потеря 1,8087 - акк: 0,3653 - val_loss: 1,6558 -
val_acc: 0,4282 Эпох а 2/40 5 с - потеря 1,5669 - акк: 0,4583 - val_loss: 1,4903 - val_acc
0,4724

...
Эпох а 40/40 5
с - потеря 0,6768 - акк: 0,7685 - val_loss: 1,2418 - val_acc: 0,5890 [INFO] оценка сети...
```

точность

вспомнить поддержку f1-score

самолет	0,62	0,68	0,65	1000
автомобиль	0,79	0,64	0,71	1000
птица	0,43	0,46	0,44	1000
Кот	0,42	0,38	0,40	1000
оленъ	0,52	0,51	0,52	1000
	0,44	0,57	0,50	1000
собака	0,74	0,61	0,67	1000
лягушка лошадь	0,71	0,61	0,66	1000
корабль	0,65	0,77	0,70	1000
грузовик	0,67	0,66	0,66	1000
среднее / общее	0,60	0,59	0,59	10000

Опять же, эпохи довольно быстрые из-за неглубокой сетевой архитектуры относительно небольшого набора данных.

Используя свой графический процессор, я получил 5-секундные эпохи, в то время как моему процессору требовалось 22 секунды для каждой эпохи.

После 40 эпох ShallowNet оценивается и выявляется, что она обеспечивает точность 60% на тестовый набор, увеличение по сравнению с предыдущей точностью 57% с использованием простых нейронных сетей.

Что еще более важно, отображение наших потерь и точности на рис. 12.3 дает нам некоторое представление о процессе обучения демонстрирует, что наши потери при проверке не растут стремительно. Наше обучение и тестирование потеря/точности начинаят расх одниться после эпохи 10. Опять же, это можно объяснить большей скоростью обучения и тот факт, что мы используем методы борьбы с переоснащением (параметры регуляризации, отсея, аугментация данных и др.).

Также известно, что набор данных CIFAR-10 легко определить из-за ограниченного количества обучающих в выборке низкого разрешения. Помимо того, как мы становимся более удобными в построении и обучении собственных пользовательских сверточных нейронных сетей, выявим методы повышения точности классификации на CIFAR-10, одновременно уменьшая переоснащение.

12.3 Резюме

В этой главе мы реализовали нашу первую архитектуру сверточной нейронной сети, ShallowNet, и обучила его на наборе данных Animals и CIFAR-10. ShallowNet получил классификацию 71%

точность на животных, увеличение на 12% по сравнению с нашим предыдущим лучшим результатом с использованием простой нейронной связи с прямой связью. сети.

Применив к CIFAR-10 точность ShallowNet достигла 60%, что выше, чем у предыдущей версии. лучший из 57% с использованием простых многослойных NN (и без значительного переобучения).

ShallowNet — чрезвычайно простая CNN, в которой используется только один слой CONV. можно получить путем обучения более глубоких сетей с несколькими наборами операций CONV => RELU => POOL.

13. Сохранение и загрузка ваших моделей

В нашей последней главе вы узнали, как обучить свою первую свёгую нейронную сеть с помощью библиотеки Keras. Однако вы возможно заметили, что каждый раз, когда вы хотите оценить свою сеть или протестировать её на наборе изображений, вам сначала нужно обучить её, прежде чем вы сможете выполнять какие-либо оценки. Это требование может быть довольно неприятно.

Мы работаем только с неглубокой сетью на небольшом наборе данных, который можно обучить относительно быстро, но что, если бы наша сеть была глубокой и нам нужно было обучить её на гораздо большем наборе данных, что заняло бы много часов или даже дней для обучения? Придется ли нам каждый раз вкладывать столько времени и ресурсов в обучение нашей сети? Или есть способ сохранить нашу модель на диск после завершения обучения, а затем просто загрузить её с диска, когда мы хотим классифицировать новые изображения?

Спорим, есть способ. Процесс сохранения загруженной модели называется сериализацией модели и является основной темой этой главы.

13.1 Сериализация модели на диск

При использовании библиотеки Keras для сериализации модели достаточно просто вызвать `model.save` для обученной модели и затем загрузить её с помощью функции `load_model`. В первой части этой главы мы изменим наш сценарий обучения ShallowNet из последней главы, чтобы сериализовать сеть после её обучения на наборе данных Animals. Затем мы создадим второй скрипт Python, который демонстрирует, как загрузить нашу сериализованную модель с диска.

Давайте начнем с обучающей части — откройте новый файл, назовите его `smallnet_train.py`, и вставьте следующий код:

```
1 # импортируем необходимые пакеты
2 из sklearn.preprocessing import LabelBinarizer 3 из
3 sklearn.model_selection import train_test_split 4 из sklearn.metrics
4 importclassification_report 5 из pyimagesearch.preprocessing import
5 ImageToArrayPreprocessor 6 из pyimagesearch.preprocessing import SimplePreprocessor
6 7 из pyimagesearch.datasets import SimpleDatasetLoader
```

```
8 из pyimagesearch.nn.conv импортировать ShallowNet 9 из
keras.optimizers импортировать SGD 10 из imutils
импортировать пути 11 импортировать matplotlib.pyplot
как plt 12 импортировать numpy как np 13 импортировать
argparse
```

Строки 2-13 импортируют необх одимые пакеты Python. Большая часть кода в этом примере идентична мелкому файлу _animals.py из главы 12. Мы рассмотрим весь файл для полноты картины и я обяжательно упомяну важные изменения внесенные для выполнения сериализации модели. Но подробный обзор того, как обучать ShallowNet на наборе данных Animals, см. в разделе 12.2.1.

Далее, давайте проанализируем наши аргументы командной строки:

```
15 # построить разбор аргумента и разобрать аргументы 16 ap =
argparse.ArgumentParser() 17 ap.add_argument("-d", "--dataset", required=True,
18     help="путь к вх одному набору
данник") 19 ap.add_argument("-m", "--model", required=True,
20     help="путь к вх одной модели") 21
args = vars(ap.parse_args())
```

В нашем предыдущем скрипте требовался только один ключ `--dataset`, который является путем к вх одному набору данных Animals. Однако, как вы можете видеть, мы добавили сюда еще один переключатель — `--model`, который указывает путь к тому месту, где мы отели бы сох ранить сеть после завершения обучения.

Теперь мы можем получить пути к изображениям в нашем `--dataset`, инициализировать наши препроцессоры и загрузите наш набор данных изображений с диска:

```
23 # получить список изображений, которые мы будем описывать 24
print("[INFO] загрузка изображений...") 25 imagePaths =
list(paths.list_images(args["dataset"]))
26
27 # инициализируем препроцессоры изображений
28 sp = SimplePreprocessor(32, 32) 29 iap =
ImageToArrayPreprocessor()
30
31 # загрузить набор данных с диска, затем масштабировать необработанные
интенсивности пикселей 32 # диапазона [0, 1] 33 sdl =
SimpleDatasetLoader(preprocessors=[sp, iap]) 34 (data, labels) = sdl.load(imagePaths, verbose
=500) 35 данные = data.astype("с плавающей запятой") / 255.0
```

Следующим шагом является разделение наших данных на обучающие и тестовые сплиты а также кодирование наших меток в виде векторов:

```
37 # разбить данные на обучающие и тестовые сплиты используя 75% из 38 # данных
для обучения и оставшиеся 25% для тестирования 39 (trainX, testX, trainY, testY) =
train_test_split(data, labels, test_size=0.25, random_state= 42)
40
41
42 # преобразовать метки из целых чисел в векторы
```

```
43 trainY = LabelBinarizer().fit_transform(trainY) 44 testY =
LabelBinarizer().fit_transform(testY)
```

Обучение ShallowNet осуществляется помошью приведенного ниже блока кода:

```
46 # инициализируем оптимизатор и модель
47 print("[INFO] компиляция модели...") 48 opt =
SGD(lr=0.005) 49 model = ShallowNet.build(width=32,
height=32, depth=3, классы=3) 50 model.compile(loss="categorical_crossentropy",
оптимизатор=opt,
51     метрики=["точность"])
52
53 # обучаем сеть 54
print("[INFO] обучая сеть...")
55 H = model.fit(trainX, trainY, validation_data=(testX, testY),
56     batch_size=32, эпох и=100, подробный=1)
```

Теперь, когда наша сеть обучена, нам нужно сохранить ее на диск. Этот процесс так же прост, как възов `model.save` и указать путь, по которому наша вък однажды должна быть сохранена на диск:

```
58 # сохранить сеть на диск
59 print("[INFO] сериализуя сеть...") 60 model.save(args["model"])
```

Метод `.save` принимает веса и состояние оптимизатора и сериализует их на диск в формате HDF5. Как мы увидим в следующем разделе, загрузить эти веса с диска так же просто, как и сохранить их.

Отсюда мы оцениваем нашу сеть:

```
62 # оценить сеть
63 print("[INFO] оценка сети...") 64 предсказания=
model.predict(testX, batch_size=32) 65 print(classification_report(testY.argmax(axis=1),
66     прогнозыargmax(ось=1),
67     target_names=["кошка", "собака", "панда"]))
```

А также график наших потерь и точности:

```
69 # график потерь и точности обучения 70
plt.style.use("ggplot") 71 plt.figure() 72
plt.plot(np.arange(0, 100), H.history["loss"], label=
"train_loss") 73 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss") 74
plt.plot(np.arange(0, 100), H.history ["acc"], label="train_acc") 75 plt.plot(np.arange(0, 100),
H.history["val_acc"], label="val_acc") 76 plt.title("Потери при обучении и Accuracy") 77
plt.xlabel("Эпох а #") 78 plt.ylabel("Потери/точность") 79 plt.legend() 80 plt.show()
```

Чтобы запустить наш скрипт, просто выполните следующую команду:

```
$ python smallnet_train.py --dataset ../datasets/animals \
--model smallnet_weights.hdf5
```

После того, как сеть закончит обучение, перечислите содержимое вашего каталога:

```
$ ls
мелкой сети_load.py мелкой сети_поезд.py мелкой сети_весов.hdf5
```

И вы увидите файл с именем `smallnet_weights.hdf5` — этот файл и есть наша сериализованная сеть. Следующий шаг — взять эту сеть и загрузить ее с диска.

13.2 Загрузка предварительно обученной модели с диска

Теперь, когда мы обучили нашу модель и сериализовали ее, нам нужно загрузить ее с диска. В качестве практического применения сериализации моделей я покажу, как классифицировать отдельные изображения из набора данных Animals, а затем отображать классифицированное изображение на нашем экране.

Откройте новый файл, назовите его `smallnet_load.py`, и мы запачкаем руки:

```
1 # импортируем необходимые пакеты
2 из pyimagesearch.preprocessing import ImageToArrayPreprocessor 3 из
pyimagesearch.preprocessing import SimplePreprocessor 4 из pyimagesearch.datasets
import SimpleDatasetLoader 5 из keras.models import load_model 6 из imutils import
paths 7 import numpy as np 8 import argparse 9 import cv2
```

Мы начинаем с импорта необходимых пакетов Python. Строки 2-4 импортируют классы, используемые для создания нашего стандартного конвейера изменения размера изображения до фиксированного размера, преобразования его в массив, совместимый с Keras, а затем с использованием этих препроцессоров для загрузки в него набора данных изображений в память.

Фактическая функция, используемая для загрузки нашей обученной модели с диска, называется `load_model` в строке 5. Эта функция ожидает за принятие пути к нашей обученной сети (файл HDF5), декодирование весов и оптимизатора внутри файла HDF5 и установку весов внутри нашей архитектуры, чтобы мы могли (1) продолжить обучение или (2) использовать сеть для классификации новых изображений.

Мы также импортируем наши привязки OpenCV в строку 9, чтобы мы могли нарисовать классификационную метку на наши изображения и отображать их на нашем экране.

Далее, давайте проанализируем наши аргументы командной строки:

```
11 # построить разбор аргумента и разобрать аргументы 12 ap =
argparse.ArgumentParser() 13 ap.add_argument("-d", "--dataset",
required=True,
14         help="путь к вх одному набору
данных") 15 ap.add_argument("-m", "--model", required=True,
16         help="путь к предварительно обученной модели")
17 args = vars(ap.parse_args())
18
19 # инициализируем метки класса
20 classLabels = ["кошка", "собака", "панда"]
```

Так же, как и в smallnet_save.py, нам потребуются два аргумента командной строки: 1. --dataset: путь к каталогу, содержащему изображения, которые мы хотим классифицировать (в данном случае набор данных Animals). 2. --model: путь к обученной сети, сериализованной на диске.

Затем в строке 20 инициализируется список меток классов для набора данных Animals.

Наш следующий блок кода обрабатывает случайную выборку десяти путей изображений из набора данных Animals для классификации:

```
22 # получить список изображений в наборе данных, а затем произвести
случайную выборку 23 # индексирует список путей к изображениям 24
print("[INFO] sample images...") 25 imagePaths =
np.array(list(paths.list_images(args[ "набор данных "]))) 26 idxs = np.random.randint(0,
len(imagePaths), size=(10,)) 27 imagePaths = imagePaths[idxs]
```

Каждое из этих десяти изображений нужно будет предварительно обработать, поэтому давайте инициализируем наши препроцессоры и загрузим десять изображений с диска:

```
29 # инициализируем препроцессоры
изображений 30 sp = SimplePreprocessor(32, 32)
31 iap = ImageToArrayPreprocessor()
32
33 # загрузить набор данных с диска, затем масштабировать необработанные
интенсивности пикселей 34 # до диапазона [0, 1] 35 sdl =
SimpleDatasetLoader(preprocessors=[sp, iap]) 36 (data, labels) = sdl.load(imagePaths)
37 данные = data.astype ("с плавающей запятой") / 255,0
```

Обратите внимание, что мы выполняем предварительную обработку изображений точно так же, как мы выполняли предварительную обработку изображений в время обучения. Невыполнение этой процедуры может привести к неправильной классификации, поскольку сети будут представлена шаблоны, которые она не может распознать. Всегда уделяйте особое внимание тому, чтобы ваши тестовые изображения были предварительно обработаны так же, как и ваши тренировочные изображения.

Далее загрузим нашу сохраненную сеть с диска:

```
39 # загрузить предварительно
обученную сеть 40 print("[INFO] загружает предварительно
обученную сеть...") 41 model = load_model(args["model"])
```

Загрузка нашей сериализованной сети так же проста, как в возврате load_model и указание пути к файлу HDF5 модели, нах однозначно диске.

Как только модель загружена, мы можем делать прогнозы для наших десяти изображений:

```
43 # делать прогнозы по изображениям 44
print("[INFO] прогнозирование...") 45 preds
= model.predict(data, batch_size=32).argmax(axis=1)
```

Имеете в виду, что метод модели .predict вернет список вероятностей для каждого изображения в данных — по одной вероятности для каждой метки класса соответственно. Беря argmax по оси = 1, мы находим индекс метки класса с наибольшей вероятностью для каждого изображения.

Теперь, когда у нас есть наши прогнозы, давайте визуализируем результаты.

```

47 # цикл по образцам изображений
48 для(i, imagePath) в перечислении (imagePaths):
49     # загрузить пример изображения, нарисовать прогноз и отобразить его
50     # на наш экран
51     изображение = cv2.imread(путь к изображению)
52     cv2.putText(изображение, "Метка: {}".format(classLabels[preds[i]]),
53                 (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
54     cv2.imshow("Изображение", изображение)
55     cv2.waitKey(0)

```

В строке 48 мы начинаем перебирать наши десять случайно выбранных путей изображения. Для каждого изображения мы загрузим его с диска (строка 51) и нарисуем предсказание метки класса на самом изображении (строки 52 и 53). В конце каждое изображение затем отображается на нашем экране в строках 54 и 55.

Чтобы попробовать smallnet_load.py, выполните следующую команду:

```
$ python smallnet_load.py --dataset ..//datasets/animals \
    --model smallnet_weights.hdf5
[INFO] образцы изображений...
[INFO] загрузка предварительно обученной сети...
[INFO] предсказание...
```

Основываясь на вышеизложенном, вы можете видеть, что наши изображения были сэмплированы предварительно обученный ShallowNet везде были загружены с диска, и что ShallowNet сделала прогнозы по нашим изображениям. Я включил или в выборку прогнозов из ShallowNet, сделанных на самих изображениях в Рисунок 13.1.



Рисунок 13.1: Пример изображений, правильно классифицированных нашей ShallowNet CNN.

Имейте в виду, что ShallowNet обеспечивает точность классификации 70% в наборе данных Animals, а это означает, что почти одно из каждого трех примеров изображений будет классифицировано неправильно. Кроме того, основываясь на classification_report из Раздела 12.2.2, мы знаем, что сеть все еще из всех сил пытается последовательно различать собак и кошек. Помимо этого, как мы продолжаем наше путешествие, применяя глубокое обучение к задачам классификации компьютерного зрения, мы рассмотрим методы, которые помогут нам повысить точность нашей классификации.

13.3 Резюме

В этой главе мы узнали, как:

Обучить сеть.

2. Сериализовать веса сети и состояние оптимизатора на диск.

3. Загрузите обученную сеть и классифицируйте изображения

Позже в главе 18 мы узнаем, как мы можем сохранить веса нашей модели на диск после каждой эпохи, что позволяет нам «проверить» нашу сеть и выбрать наиболее эффективную. Сохранение весов модели в время фактического процесса обучения также позволяет нам перезапустить обучение с определенной точки, если наша сеть начинает демонстрировать признаки переобучения. Процесс остановки обучения настройки параметров и повторного запуска обучения подробно описан в пакетах Practitioner Bundle и ImageNet Bundle.

14. LeNet: распознавание рукописных цифр

Архитектура LeNet — это основополагающая работа в сообществе глубокого обучения в первые представления LeCun et al. в своей статье 1998 года «Обучение на основе градиента, применяемое к распознаванию документов» [19]. Как следует из названия документа, мотивация авторов при внедрении LeNet была в первую очередь связана с оптическим распознаванием символов (OCR).

Архитектура LeNet проста и мала (с точки зрения занимаемой памяти), что делает ее идеально подходящей для обучения основам CNN.

В этой главе мы попытаемся воспроизвести эксперименты, подобные экспериментам Лекуна в их статье 1998 года. Мы начнем с обзора архитектуры LeNet, а затем реализуем ее с помощью Keras. Наконец, мы оценим LeNet в наборе данных MNIST для распознавания рукописных цифр.

14.1 Архитектура LeNet

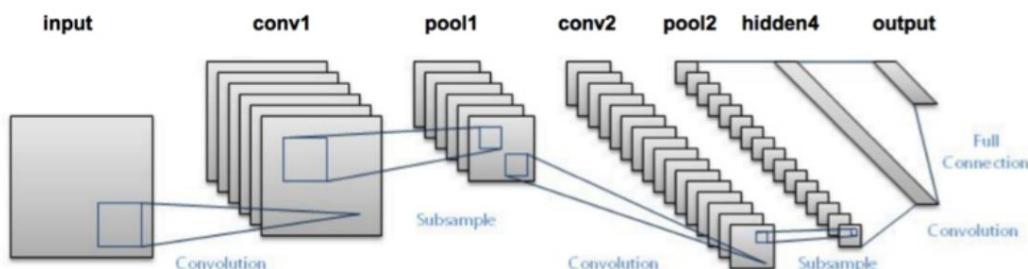


Рисунок 14.1: Архитектура LeNet состоит из двух серий наборов слоев CONV => TANH => POOL, за которыми следуют полно связанный слой и вывод softmax. Фото предоставлено <http://pyimg.co/lhsx>

Теперь, когда мы изучили строительные блоки сверточных нейронных сетей в главе 12 с использованием ShallowNet, мы готовы сделать следующий шаг и обсудить LeNet. Архитектура LeNet

Тип слоя	Възможности одноготома	размер фильтра / Шаг
ВХ ОДНОЕ ИЗОБРАЖЕНИЕ	28×28×1	
КОНВ.	28×28×20 5×5, K = 20	
ДЕЙСТВИЕ	28×28×20	
БАССЕЙН	14×14×20 2×2	
КОНВ.	14×14×50 5×5, K = 50	
ДЕЙСТВИЕ	14×14×50	
БАССЕЙН	7×7×50 2×2	
ФК	500	
ДЕЙСТВИЕ	500	
ФК	10	
СОФТМАКС	10	

Таблица 14.1: Своя таблица архитектуры LeNet. Размеры възможности одноготома включены для каждого слоя, а также размер сверточного фильтра/размер пула, когда это уместно.

(рис. 14.1) — отличная первая «реальная» сеть. Сеть небольшая и простая для понимания

— но достаточно большой, чтобы обеспечить интересные результаты

Кроме того, комбинация LeNet + MNIST может легко работать на ЦП, что делает новичкам легкодоступной свой первый шаг в глубоком обучении и CNN. Всюду LeNet + MNIST — это аналог «Hello, World» глубокого обучения применяемой классификации изображений. Архитектура LeNet состоит из следующих слоев, использующих шаблон CONV => ACT => POOL .

из раздела 11.3:

INPUT => CONV => TANH => POOL => CONV => TANH => POOL =>
ФК => TANH => ФК

Обратите внимание, как в архитектуре LeNet используется функция активации tanh, а не более популярная ReLU. Еще в 1998 году ReLU не использовалась в контексте глубокого обучения — он был более обычным и использовал tanh или сигмоиду в качестве функции активации. При введении LeNet сегодня важно обычное заменить TANH на ReLU — мы будем следовать этому же правилу и использовать ReLU в качестве нашего функции активации далее в этой главе.

Таблица 14.1 суммирует параметры архитектуры LeNet. Наш слой принимает ввод изображение с 28 строками, 28 столбцами и одним каналом (оттенки серого) для глубины (т. е. размеры изображения внутри набора данных MNIST). Затем мы изучаем 20 фильтров, каждый из которых имеет размер 5×5. КОНВ — за слоем следует активация ReLU, за которой следует максимальное объединение с размером 2×2 и шагом 2×2 .

Следующий блок архитектуры следует той же схеме, на этот раз изучая 50 фильтров 5×5 . Обычное количество слоев CONV увеличивается более глубоких слоев сети, поскольку фактическое пространство възможностей размеров уменьшается

Затем у нас есть два слоя FC . Первый FC содержит 500 скрытых узлов, за которыми следует ReLU. Активация последний слой FC управляет количеством меток възможности одного класса (0-9; по одной для каждого из возможных десяти цифр). Наконец, мы применяем активацию softmax для получения вероятностей класса.

14.2 Введение LeNet

Учитывая Таблицу 14.1 выше, мы теперь готовы реализовать исходную архитектуру LeNet, используя библиотеку Keras. Начните с добавления нового файла с именем lenet.py в папку pyimagesearch.nn.conv . sub-module — в этом файле будет храниться наша реальная реализация LeNet:

```
-- pyimagesearch
| |--- __init__.py
| |--- __init__.py
| |--- __init__.py
...
| |--- конв.
| |--- __init__.py
| |--- lenet.py
| |--- мелкаясеть.py
```

Оттуда откройте lenet.py, и мы можем начать программировать:

```
1 # импортируем необх одимък пакеты
2 из импорта keras.models Последовательный
3 из keras.layers.convolutional импорт Conv2D
4 из keras.layers.convolutional импортировать MaxPooling2D
5 из keras.layers.core импорт Активация
6 из keras.layers.core import Flatten
7 из keras.layers.core иморт Плотный
8 из keras импортировать бэкэнд как K
```

Строки 2-8 обрабатывают импорт наших необх одимък пакетов Python — именно это импорт такие же, как реализация ShallowNet из главы 12, и формируют необх одимък набор требуемых импортирует при построении (почти) любой CNN с использованием Keras.

Затем мы определяем метод сборки LeNet ниже, используемый для фактического построения сети. архитектура:

10 класс Ленет:

```
11     @staticmethod
12     def build(ширина, высота, глубина, классы):
13         # инициализируем модель
14         модель = Последовательный()
15         inputShape = (высота, ширина, глубина)
16
17         # если мы используем "сначала каналы", обновите форму ввода
18         если K.image_data_format() == "channels_first":
19             inputShape = (глубина, высота, ширина)
```

Метод сборки требует четырех параметров:

1. Ширина вх одного изображения
2. Высота вх одного изображения
3. Количество каналов (глубина) изображения
4. Числометок класса в задаче классификации.

Класс Sequential, строительный блок последовательных сетей, последовательно складывает один слой. поверх другого инициализируется в строке 14. Затем мы инициализируем inputShape, как если бы использовали «каналы последний» заказ. В случае, если наша конфигурация Keras настроена на использование порядка «каналы в первую очередь», мы обновить inputShape в строках 18 и 19.

Первый набор слоев CONV => RELU => POOL определен ниже:

```

21         # первый набор слоев CONV => RELU => POOL
22         model.add(Conv2D(20, (5, 5), padding="тоже самое",
23                         input_shape=inputShape))
24         model.add(Активация("relu"))
25         model.add(MaxPooling2D(pool_size=(2, 2), шаги=(2, 2)))

```

Наш слой CONV изучит 20 фильтров, каждый размером 5×5 . Затем мы применяем активацию ReLU, функция за которой следует объединение 2×2 с шагом 2×2 , тем самым уменьшая размер вх одноготома на 75%.

Затем применяется другой набор слоев CONV => RELU => POOL, на этот раз изучая 50 фильтров . а не 20:

```

27         # второй набор слоев CONV => RELU => POOL
28         model.add(Conv2D(50, (5, 5), padding="тоже самое",
29                         input_shape=inputShape))
30         model.add(Активация("relu"))
31         model.add(MaxPooling2D(pool_size=(2, 2), шаги=(2, 2)))

```

Затем вх одногом можно сгладить и получить полно связанный слой с 500 узлами.

применимый:

```

32         # первый (и единственный) набор слоев FC => RELU
33         model.add(Свести())
34         model.add(плотный (500))
35         model.add(Активация("relu"))

```

Затем следует окончательный классификатор softmax:

```

37         # классификатор softmax
38         model.add(плотный (классы))
39         model.add(Активация("softmax"))
40
41         # вернуть построенную сетевую архитектуру
42         модель возврата

```

Теперь, когда мы скодировали архитектуру LeNet, мы можем перейти к ее применению к MNIST. набор данных .

14.3 LeNet на MNIST

Следующим шагом будет создание скрипта драйвера, отвечающего:

1. Загрузка набора данных MNIST с диска.
2. Создание экземпляра архитектуры LeNet.
3. Обучение LeNet.
4. Оценка производительности сети.

Чтобы обучить и оценить LeNet на MNIST, создайте новый файл с именем lenet_mnist.py, и мы можем начать:

¹ # импортируем необходимые пакеты

² из pyimagesearch.nn.conv и import LeNet

```
3 из keras.optimizers импортировать SGD 4
из sklearn.preprocessing импортировать LabelBinarizer 5 из
sklearn.model_selection импортировать train_test_split 6 из
sklearn.metrics импортировать Classification_report 7 из sklearn
импортировать наборы данных 8 из keras импортировать
серверную часть как K 9 импортировать matplotlib.pyplot как plt
10 импортировать numpy как np
```

На этом этапе наши import Python должен начать казаться довольно стандартным с заметным шаблоном. появление. В подавляющем большинстве примеров в этой книге нам придется импортировать:

1. Сетевая архитектура, которую мы будем обучать.
2. Оптимизатор для обучения сетей (в данном случае SGD).
3. (Набор) удобных функций, используемых для построения тренировочного и тестового разделения заданного набора данных.
4. Функция для расчета отчета о классификации, чтобы мы могли оценить производительность нашего классификатора.

Опять же, почти все примеры этой книги будут следовать этому шаблону импорта, наряду с некоторыми дополнительными классами здесь и там для облегчения определенных задач (таких как предварительная обработка изображений). Набор данных MNIST уже был предварительно обработан, поэтому мы можем просто загрузить его с помощью следующего ввода функции:

```
12 # получить набор данных MNIST (если вы впервые используете этот набор
данных 13 #, загрузка 55 МБ может занять минуту) 14 print("[INFO] доступ к MNIST...")
15 dataset = datasets.fetch_mldata("MNIST_Оригинал") 16 данные = набор данных .данные
```

Строка 15 загружает набор данных MNIST с диска. Если вы впервые вызываете функцию fetch_mldata с строкой «MNIST Original», набор данных MNIST необязательно загрузить из репозитория наборов данных mldata.org. Набор данных MNIST сериализован в один файл размером 55 МБ, поэтому в зависимости от вашего интернет-соединения эта загрузка может занять от нескольких секунд до нескольких минут.

Важно отметить, что каждая выборка MNIST внутри данных представлена 784-дневным вектором (т. е. интенсивностью необработанных пикселей) изображения в градациях серого 28×28. Следовательно, нам нужно изменить матрицу данных в зависимости от того, используем ли мы порядок «каналы сначала» или «каналы в последнюю очередь»:

```
18 # если мы используем упорядочение "сначала каналы", то измените формат
матрицы дизайна 19 # таким образом, чтобы матрица была следующей: 20 #
num_samples x глубина x строк x столбцов 21 if K.image_data_format() ==
"channels_first":
22     данные = данные.изменить(формат[0], 1, 28, 28)
23
24 # в противном случае мы используем порядок «каналы в последнюю очередь»,
поэтому дизайн 25 # форма матрицы должна быть: num_samples x строк x столбцов x
глубина 26 else: data = data.reshape(data.shape[0], 28, 28, 1)
27
```

Если мы выполняем упорядочивание «сначала каналы» (строки 21 и 22), то матрица данных изменяется таким образом, что количество в выборке является первой записью в матрице, один канал — второй записью, за которой следует количество строк и столбцов (28 и 28 соответственно). В противном случае мы предполагаем, что используем порядок «каналы последними», и в этом случае матрица преобразуется в число каналов.

сначала в выборки, количество строк, количество столбцов и, наконец, количество каналов (строки 26 и 27).

Теперь, когда наша матрица данных имеет правильную форму, мы можем выполнить разделение обучения и тестирования в звя сначала позаботьтесь о масштабировании интенсивности пикселей изображения до диапазона [0,1]:

```
29 # масштабируем вх одные данные до диапазона [0, 1] и выполним обучение/
тест 30 # разделение 31 (trainX, testX, trainY, testY) = train_test_split(data / 255.0,
```

```
32     набор данных .target.astype («целое число»), test_size = 0.25, random_state = 42)
33
34 # преобразовать метки из целых чисел в векторы 35 le = LabelBinarizer() 36
trainY = le.fit_transform(trainY) 37 testY = le.transform(testY)
```

После разделения данных мы также кодируем наши метки классов как горячие векторы, а не отдельные целевые значения. Например, если метка класса для данной выборки равна 3, то результат прямого кодирования метки будет следующим: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

Обратите внимание, что все записи в векторе равны нулю, за исключением четвертого индекса, который теперь равен единице. (имейте в виду, что цифра 0 — это первый индекс, поэтому три — это четвертый индекс).

Теперь все готово для обучения LeNet на MNIST:

```
39 # инициализируем оптимизатор и модель
40 print("[INFO] компилируем модель...") 41
opt = SGD(lr=0.01) 42 model = LeNet.build(width=28,
height=28, depth=1, классы=10) 43 model.compile(loss="categorical_crossentropy",
оптимизатор=opt,
44     метрики=["точность"])
45
46 # обучаем сеть
47 print("[INFO] обучаю щасеть...")
48 H = model.fit(trainX, trainY, validation_data=(testX, testY),
49     batch_size=128, эпох и=20, подобный=1)
```

Строка 41 инициализирует наш оптимизатор SGD со скоростью обучения 0,01. Сам LeNet создается в строке 42, что указывает на то, что все вх одные изображения в нашем наборе данных будут иметь ширину 28 пикселей, высоту 28 пикселей и глубину 1. Учитывая что в наборе данных MNIST имеется десять классов (по одному для каждой цифры, 0–9), мы устанавливаем классы=10.

Строки 43 и 44 компилируют модель, используя кросс-энтропийную потерю в качестве нашей функции потери. Страна 48 и 49 обучает LeNet на MNIST в общей сложности 20 эпох, используя размер мини-пакета 128.

Наконец, мы можем оценить производительность нашей сети, а также построить график потерь и точности. Современем в последнем блоке кода ниже:

```
51 # оценить сеть
52 print("[ИНФО] оценка сети...") 53
предсказания = model.predict(testX, batch_size=128) 54
print(classification_report(testY.argmax(axis=1),
55     прогнозы.argmax(ось=1),
56     target_names=[str(x) для x в le.classes_]))
57
```

```

58 # график потери и точности обучения
59 plt.style.use("ggplot")
60 plt.цифра()
61 plt.plot(np.arange(0, 20), H.history["потеря"], метка="train_loss")
62 plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
63 plt.plot(np.arange(0, 20), H.history["acc"], label="train_acc")
64 plt.plot(np.arange(0, 20), H.history["val_acc"], label="val_acc")
65 plt.title("Тренировочное потери и точность")
66 plt.xlabel("Эпох а #")
67 plt.ylabel("Потери/точность")
68 plt.легенда()
69 plt.показать()

```

Я упоминал об этом ранее в Разделе 12.2.2 при оценке ShallowNet, но убедитесь, что вы поняли, что делает строка 53, когда в вызывается `model.predict`. Для каждого образца в `testX` создаются партии размером 128 штук, которые затем передаются сеть для классификации. После этого точки данных тестирования были классифицированы в зависимости от предсказанных прогнозов.

Переменная `прогнозов` на самом деле представляет собой массив NumPy формы `(len(testX), 10)`, подразумевая, что теперь у нас есть 10 вероятностей, связанных с каждой меткой класса для всех данных точек в `testX`. Получение прогнозов `.argmax` (ось = 1) в `classification_report` на линиях 54-56 находит индекс метки с наибольшей вероятностью (т.е. окончательную вью одноклассификацию). Учитывая окончательную классификацию из сети, мы можем сравнить наши предсказанные метки классов с этикетками достоверной информации.

Чтобы выполнить наш скрипт, просто введите следующую команду:

```
$ питон lenet_mnist.py
```

Затем следует загрузить набор данных MNIST и/или загрузить его с диска, после чего следует провести обучение. начать:

```

[INFO] доступ к MNIST...
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 52 500 образцах, проверка на 17 500 образцах
Эпох а 1/20
3s - убыток: 1,0970 - акк: 0,6976 - val_loss: 0,5348 - val_acc: 0,8228
...
Эпох а 20/20
3 с - убыток: 0,0411 - акк: 0,9877 - знач_убыток: 0,0576 - знач_акк: 0,9837
[INFO] оценка сети...

```

	точность	вспомнить поддержку	f1-score	
0	0,99	0,99	0,99	1677
1	0,99	0,99	0,99	1935 г.
2	0,99	0,98	0,99	1767 г.
3	0,99	0,97	0,98	1766
4	1,00	0,98	0,99	1691
5	0,99	0,98	0,98	1653
6	0,99	0,99	0,99	1754 г.
7	0,98	0,99	0,99	1846 г.
8	0,94	0,99	0,97	1702 г.
9	0,98	0,98	0,98	1709

среднее / общее	0,98	0,98	17500
-----------------	------	------	-------

Используя графический процессор Titan X, я получал трех секундные эпохи. Используются только ЦП, число секунд в эпохах подскочило до тридцати. После завершения обучения видим, что LeNet получает точность классификации 98%, огромный рост по сравнению с 92% при использовании стандартной нейронной сети с прямой связью. Сеть в главе 10.

Более того, если посмотреть на график потерь и точности во времени на рис. 14.2, видно, что наша сеть ведет себя достаточно хорошо. Спустя всего пять эпох LeNet уже достигает 96% точности классификации. Потери данных как для обучения так и для проверки продолжают снижаться и несколько незначительных «всплесков» из-за того, что наша скорость обучения остается постоянной и не затухает (концепция мы рассмотрим позже в главе 16). В конце двадцати эпох мы достигаем точности 98% на нашем тестовом наборе.



Рисунок 14.2: Обучение LeNet работе с MNIST. Всего через двадцать эпох мы получаем точность классификации 98%.

Этот график, демонстрирующий потери и точность LeNet на MNIST, возможно, является квинтэссенцией графика, который мы имеем: потери и точность при обучении и проверке тесно взаимодействуют друг с другом (почти) без признаков переобучения. Как мы видим, часто бывает очень трудно пройти этот тип обучения: график, который ведет себя так хорошо, указывая на то, что наша сеть изучает базовые шаблоны без переоснащения.

Существует также проблема, заключающаяся в том, что набор данных MNIST сильно предварительно обработан и не является представительным для проблем классификации изображений, с которыми мы столкнемся в реальном мире. Исследователи склонны использовать набор данных MNIST в качестве эталона для оценки новых алгоритмов классификации. Если их методы могут получить > 95% точности классификации, то есть либо ошибка в (1) логике алгоритма, либо

(2) сама реализация

Тем не менее, применение LeNet к MNIST — отличный способ получить первый опыт применения глубокого обучения к задачам классификации изображений и имитировать результаты оригинального исследования LeCun et al. бумаги.

14.4 Резюме

В этой главе мы рассмотрели архитектуру LeNet, представленную LeCun et al. в своей статье 1998 года «Обучение на основе градиента, применяемое к распознаванию документов» [19]. LeNet — основополагающая работа в литературе по глубокому обучению — она подробно продемонстрировала, как можно научить нейронные сети распознавать объекты на изображениях сквозным образом (т. е. не нужно было извлекать признаки, сеть могла обучаться узорами из самих изображений).

Несмотря на то, что LeNet является оригинальной сетью, посегодняшним меркам она по-прежнему считается «мелкой» сетью. Имея всего четыре обучаемых слоя (два слоя CONV и два слоя FC), глубина LeNet меркнет по сравнению с глубиной современных современных архитектур, таких как VGG (16 и 19 слоев) и ResNet (более 100 слоев).).

В нашей следующей главе мы обсудим вариант архитектуры VGGNet, который я называю «*MiniVGGNet*». В этом варианте архитектуры используются те же руководящие принципы, что и в работе Симона и Зиссермана [95], но меняется глубина, что позволяет нам обучать сеть на меньших наборах данных. Для полной реализации архитектуры VGGNet вы можете обратиться к Главе 6 комплекта ImageNet, где мы обучаем VGGNet с нуля на ImageNet.

15. MiniVGGNet: углубляясь в CNN

В нашей предыдущей главе мы обсудили LeNet, оригинальную сверточную нейронную сеть в литературе по глубокому обучению и компьютерному зрению. VGGNet (иногда называемая просто VGG) была впервые представлена Симоном и Зиссерманом в их статье 2014 года «Сверточные нейронные сети с очень глубоким обучением для крупномасштабного распознавания изображений» [95]. Основой вклад их работы заключался в том, чтобы продемонстрировать, что архитектуру с очень маленькими (3×3) фильтрами можно обучить на все большей глубине (16-19 слоев) и получить современную классификацию в сложной задаче классификации ImageNet.

Ранее сетевые архитектуры в литературе по глубокому обучению использовали фильтры разных размеров: первый уровень CNN обычно включает размеры фильтров где-то между 7×7 [94] и 11×11 [128]. Оттуда размеры фильтров постепенно уменьшались до 5×5 . Наконец, только самые глубокие слои сети использовали фильтры 3×3 .

VGGNet уникален тем, что использует ядра 3×3 во всей архитектуре. Использование этих небольших ядер, возможно, помогает VGGNet обобщать проблемы классификации за пределами того, на чем изначально обучалась сеть (мы видим это в пакетах Practitioner Bundle и ImageNet Bundle, когда будем обсуждать трансферное обучение).

Каждый раз, когда вы видите сетевую архитектуру, полностью состоящую из фильтров 3×3 , вы можете быть уверены, что она была вдохновлена VGGNet. Обзор всех 16- и 19-уровневых вариантов VGGNet слишком сложен для этого введения в сверточные нейронные сети — для подробного обзора VGG16 и VGG19, см. главу 11 комплекта ImageNet.

Вместо этого мы рассмотрим семейство сетей VGG и определим, какие характеристики должна иметь CNN, чтобы вписаться в это семейство. Оттуда мы реализуем уменьшенную версию VGGNet под названием MiniVGGNet, которую можно легко обучить в вашей системе. Эта реализация также продемонстрирует, как использовать два важных уровня, которые мы обсуждали в главе 11 — нормализацию партии (BN) и отсев.

15.1 Семейство сетей VGG

Семейство сверточных нейронных сетей VGG можно охарактеризовать двумя ключевыми компонентами:

1. Все слои CONV в сети используют только фильтры 3×3 .

2. Наложение нескольких наборов слоев CONV => RELU (где количество о последовательных CONV => Уровни RELU обычно увеличиваются по мере того, как мы идем глубже) перед применением операции POOL.

В этом разделе мы собираемся обсудить вариант архитектуры VGGNet, который называется «MiniVGGNet» из-за того, что есть существенно более мелкая, чем ее старший брат. Подробный обзор и реализация оригинальной архитектуры VGG, предложенной Симоном и Зиссерманом, наряду с демонстрацией обучения сети на наборе данных ImageNet, см. главу 11 комплекта ImageNet.

15.1.1 Архитектура (мини) VGGNet

И в ShallowNet, и в LeNet мы применили серию слоев CONV => RELU => POOL. Однако в VGGNet мы объединяем несколько слоев CONV => RELU перед применением одного слоя POOL.

Это позволяет сети изучать более богатые функции из слоев CONV до субдискретизации пространственного в ходного размера с помощью операции POOL.

В целом, MiniVGGNet состоит из двух наборов CONV => RELU => CONV => RELU => POOL. слоев, за которыми следует набор слоев FC => RELU => FC => SOFTMAX. Первые два слоя CONV включают 32 фильтра, каждый размером 3x3. Вторые два слоя CONV изучают 64 фильтра, опять же, каждый размером 3x3. Наши слои POOL будут включать максимальное объединение в окне 2x2 с шагом 2x2. Мы также будем вставлять слои пакетной нормализации после активации вместе со слоями исключения (DO) после слоев POOL и FC.

Сама сетевая архитектура подробно описана в таблице 15.1, где начальный размер в ходного изображения равен. Предполагается, что это 32x32x3, так как мы будем обучать MiniVGGNet на CIFAR-10 позже в этой главе. (а затем сравнение производительности с ShallowNet).

Опять же, обратите внимание, как слои пакетной нормализации и отсева включены в сеть. архитектура, основанная на моих «эмпирических правилах» в разделе 11.3.2. Применение пакетной нормализации поможет уменьшить влияние переобучения и повысить точность нашей классификации на CIFAR-10.

15.2 Внедрение MiniVGGNet

Учитывая описание MiniVGGNet в таблице 15.1, теперь мы можем реализовать сетевую архитектуру с помощью Keras. Для начала добавьте новый файл с именем minivggnet.py в файл pyimagesearch.nn.conv. sub-module — там мы напишем нашу реализацию MiniVGGNet:

```
-- pyimagesearch
| |--- __init__.py
| |--- nn
| | |--- __init__.py
...
| | |--- CONV.
| | |--- __init__.py
| | |--- lenet.py
| | |--- minivggnet.py
| | |--- мелкаясеть.py
```

После создания файла minivggnet.py откройте его в своем любимом редакторе кода, и мы перейдем к работе:

```
1 # импортируем необходимые пакеты
2 из импорта keras.models Последовательный
3 из keras.layers.normalization import BatchNormalization
4 из keras.layers.convolutional импорт Conv2D
```

Тип слоя	Вък одной размер	Размер фильтра/шаг
ВХОДНОЕ ИЗОБРАЖЕНИЕ	32×32×3	
КОНВ.	32×32×32 3×3, K = 32	
ДЕЙСТВИЯ	32×32×32	
БН	32×32×32	
КОНВ.	32×32×32 3×3, K = 32	
ДЕЙСТВИЯ	32×32×32	
БН	32×32×32	
БАССЕЙН	16×16×32 2×2	
ВЫБЫВАТЬ	16×16×32	
КОНВ.	16×16×64 3×3, K = 64	
ДЕЙСТВИЯ	16×16×64	
БН	16×16×64	
КОНВ.	16×16×64 3×3, K = 64	
ДЕЙСТВИЯ	16×16×64	
БН	16×16×64	
БАССЕЙН	8×8×64 2×2	
ВЫБЫВАТЬ	8×8×64	
ФК	512	
ДЕЙСТВИЯ	512	
БН	512	
ВЫБЫВАТЬ	512	
ФК	10	
СОФТМАКС	10	

Таблица 15.1: Структура архитектуры MiniVGGNet. Размеры вък одного тома включены для каждого слоя, а также размер сверточного фильтра/размер ядра, если это необязательно. Обратите внимание, что только 3×3 применяется к сверткам.

```

5 из keras.layers.convolutional импортировать MaxPooling2D
6 из keras.layers.core импорт Активация
7 из keras.layers.core import Flatten
8 из keras.layers.core import Dropout
9 из keras.layers.core импорт Плотный
10 из keras импортировать бэкенд как K

```

Строки 2-10 импортируют необх одимье нам классы из библиотеки Keras. Большая часть этого импорта у вас есть уже в видели раньше, но я очу обратить в аще в внимание на BatchNormalization (строка 3) и Отсев (строка 8) — эти классы позволят нам применить пакетную нормализацию и отсев к нашим сетевая архитектуре.

Как и в наших реализациях ShallowNet и LeNet, мы определим метод сборки , который может быть введен для построения архитектуры с использованием предоставленной ширины высоты глубины количества классов

12 класс МиниVGGNet:

```

13     @статический метод
14     def build(ширина, высота, глубина, классы):
15         # инициализируем модель вместе с вх одной формой, чтобы она была
16         # "каналы последние" и самоизмерение каналов
17         модель = Последовательный()
18         inputShape = (высота, ширина, глубина)
19         ЧанДим = -1
20
21         # если мы используем "сначала каналы", обновите форму ввода
22         # и размер каналов
23         если K.image_data_format() == "channels_first":
24             inputShape = (глубина, высота, ширина)
25             ЧанДим = 1

```

Строка 17 создает экземпляр класса Sequential , строительного блока последовательных нейронных сетей в Керас. Затем мы инициализируем inputShape , предполагая что мы используем последний порядок каналов (строка 18).

В строке 19 представлена переменная которую мы раньше не видели, chanDim , индекс канала. измерение. Пакетная нормализация работает по каналам, поэтому, чтобы применить BN , нам нужно знать, по какой оси нормализовать. Установка chanDim = -1 подразумевает, что индекс канала измерение последним в вх одной форме (т. е. каналы в последнем порядке). Однако, если мы используем каналы первый заказ (строки 23-25), нам нужно обновить inputShape и установить chanDim = 1, так как размер канала теперь является первой записью в вх одной форме.

Блок первого уровня MiniVGGNet определен ниже:

```

27     # первый набор слоев CONV => RELU => CONV => RELU => POOL
28     model.add(Conv2D(32, (3, 3), padding="тоже самое",
29                     input_shape=inputShape))
30     model.add(Активация("relu"))
31     model.add(Пакетная нормализация(ось = chanDim))
32     model.add(Conv2D(32, (3, 3), padding="тоже самое"))
33     model.add(Активация("relu"))
34     model.add(Пакетная нормализация(ось = chanDim))
35     model.add(MaxPooling2D(pool_size=(2, 2)))
36     model.add(Впадение (0,25))

```

Здесь мы видим, что наша архитектура состоит из (CONV => RELU => BN) * 2 => POOL => DO.

Строка 28 определяет слой CONV с 32 фильтрами, каждый из которых имеет размер фильтра 3x3. Затем мы применяем АктивациюReLU (строка 30), которая немедленно передается на уровень пакетной нормализации (строка 31). Для нулевого центра активаций.

Однако вместо этого, чтобы применять слой POOL для уменьшения пространственных размеров нашего ввода, мы вместо этого примените другой набор CONV => RELU => BN — это позволит нашей сети узнать больше особенностей, обычной практика при обучении более глубоких CNN.

В строке 35 мы используем MaxPooling2D размером 2x2. Поскольку мы явно не задаем шаг, Keras неявно предполагает, что наш шаг равен максимальному размеру пула (который равен 2×2).

Затем мы применяем Dropout к строке 36 с вероятностью $p = 0,25$, что означает, что узел из слоя POOL будет случайным образом отключаться от следующего слоя с вероятностью 25% вовремя обучения. Мы применяем отсев, чтобы уменьшить влияние переобучения. Вы можете прочитать больше об исключениях в разделе 11.2.7. Затем мы добавляем блок второго уровня в MiniVGGNet ниже:

```

38      # второй набор слоев CONV => RELU => CONV => RELU => POOL
39      model.add(Conv2D(64, (3, 3), padding="тоже самое"))
40      model.add(Активация("relu"))
41      model.add(Пакетная нормализация(ось = chanDim))
42      model.add(Conv2D(64, (3, 3), padding="тоже самое"))
43      model.add(Активация("relu"))
44      model.add(Пакетная нормализация(ось = chanDim))
45      model.add(MaxPooling2D(pool_size=(2, 2)))
46      model.add(Выпадение (0,25))

```

Приведенный выше код следует той же схеме, что и выше; однако сейчас мы изучаем два набора из 64 фильтров (каждый размером 3x3) вместо 32 фильтров. Статья же, обычно увеличиваются количество фильтров помимо того, как размер пространственного ввода уменьшается глубже в сети.

Далее идет наш первый (и единственный) набор слоев FC => RELU:

```

48      # первый (и единственный) набор слоев FC => RELU
49      model.add(Свести())
50      model.add(глобальный (512))
51      model.add(Активация("relu"))
52      model.add(Пакетная нормализация())
53      model.add(Выпадение (0,5))

```

Наш уровень FC имеет 512 узлов, за которыми следует активацияReLU и BN. Хорошо также примените здесь отсев, увеличив вероятность до 50% — обычно вы видите отсев с $p = 0,5$ применяется между слоями FC.

Наконец, мы применяем классификатор softmax и возвращаем сетевую архитектуру в вьюшней стороне функция

```

55      # классификатор softmax
56      model.add(глобальный (классы))
57      model.add(Активация("softmax"))

58
59      # вернуть построенную сетевую архитектуру
60      модель возврата

```

Теперь, когда мы реализовали архитектуру MiniVGGNet, давайте перейдем к ее применению к CIFAR-10.

15.3 MiniVGGNet на CIFAR-10

Мы будем следовать той же схеме обучения MiniVGGNet, что и для LeNet в главе 14, только на этот раз с набором данных CIFAR-10:

- Загрузите набор данных CIFAR-10 с диска.
- Создание экземпляра архитектуры MiniVGGNet.
- Обучите MiniVGGNet, используя обучение данные.

Оцените производительность сети с помощью данных тестирования

Чтобы создать скрипт драйвера для обучения MiniVGGNet, откройте новый файл, назовите его `minivggnet_cifar10.py`, и вставьте следующий код:

```
1 # установить бэкэнд matplotlib, чтобы оно можно было со временем 2
import matplotlib 3 matplotlib.use("Agg")

4
5 # импортируем необходимые пакеты
из sklearn.preprocessing импортируем LabelBinarizer 7 из
sklearn.metrics импортируем classification_report 8 из
pyimagesearch.nn.conv импортируем MiniVGGNet 9 из
keras.optimizers импортируем SGD 10 из keras.datasets
импортируем cifar10 11 импортируем matplotlib.pyplot как plt 12
импортировать numpy как np 13 импортировать argparse
```

Строка 2 импортирует библиотеку `matplotlib`, которую мы можем использовать для построения графиков точности и потерь в времени. Нам нужно установить для бэкенда `matplotlib` значение `Agg`, чтобы указать, что нужно создать нерактивный объект, который будет просто со временем на диске. В зависимости от вашего бэкенда `matplotlib` по умолчанию и от того, получаете ли вы удаленный доступ к своей машине глубокого обучения (например, через SSH), сеанс X11 может истечь потаймаутом. Если это произойдет, `matplotlib` выдаст ошибку при попытке отобразить вашу фигуру. Вместо этого мы можем просто установить флаг `Agg` и записать график на диск, когда мы закончим обучение нашей сети.

Строки 9-13 импортируют остальные наши необходимые пакеты Python, все из которых мы видели раньше — исключением является MiniVGGNet на линии 11, которую мы реализовали в предыдущем разделе.

Далее, давайте проанализируем наши аргументы командной строки:

```
15 # построить разбор аргумента и разобрать аргументы 16 ap =
argparse.ArgumentParser() 17 ap.add_argument("-o", "--output",
required=True,
18         help="путь к выходному графику потерь/точности") 19
args = vars(ap.parse_args())
```

Этот скрипт потребует только один аргумент командной строки, `--output`, путь к нашему выходу сюжет тренировки и прогресса.

Теперь мы можем загрузить набор данных CIFAR-10 (предварительно разделенный на обучение и тестовые данные), масштабировать пиксели в диапазоне [0,1], а затем сразу закодировать метки:

```
21 # загрузить данные обучения и тестирования, затем масштабировать их в диапазоне 22 # диапазон [0, 1] 23 print("[INFO]
загрузка данных CIFAR-10...") 24 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
```

```

25 trainX = trainX.astype("float") / 255.0 26 testX = testX.astype("float") /
255.0
27
28 # преобразуем метки из целых чисел в векторы 29 lb = LabelBinarizer()
30 trainY = lb.fit_transform(trainY) 31 testY = lb.transform(testY)

32
33 # инициализировать имена меток для набора данных CIFAR-10
34 labelNames = ["самолет", "автомобиль", "птица", "кошка", "олень", "собака", "лягушка", "лошадь",
35     "корабль", "грузовик"]

```

Скомпилируем нашу модель и начнем обучение MiniVGGNet:

```

37 # инициализируем оптимизатор и модель 38 print("[INFO]")
компилируем модель...") 39 opt = SGD(lr=0.01, затухание=0.01 /
40, импульс=0.9, нестреков=True) 40 model = MiniVGGNet.build(ширина = 32, высота = 32, глубина = 3,
классы = 10) 41 model.compile(loss="categorical_crossentropy", оптимизатор=opt,
42     метрики=["точность"])
43
44 # обучаем сеть
45 print("[INFO] обучая цепочку...")
46 H = model.fit(trainX, trainY, validation_data=(testX, testY),
47     batch_size=64, эпохи=40, подробности=1)

```

Мы будем использовать SGD в качестве нашего оптимизатора со скоростью обучения = 0,1 и импульсом = 0,9. Параметр Nestrov=True указывает, чтобы мы могли применить ускоренный градиент Нестровова к оптимизатору SGD (раздел 9.3).

Термин оптимизатора, который мы не видели, — это параметр затухания. Этот аргумент используется для медленного снижения скорости обучения со временем. Как мы уже обсудили в следующей главе о планировании скорости обучения, снижение скорости обучения помогает уменьшить переобучение и получить более высокую точность классификации — чем меньше скорость обучения тем меньше будут обновления веса. Обычно настройка для затухания — разделить начальную скорость обучения на общее количество эпох — в этом случае мы будем обучать нашу сеть в общей сложности 40 эпох с начальной скоростью обучения 0,01, поэтому затухание = 0,01/40.

После завершения обучения мы можем оценить сеть и отобразить хорошо отформатированную классификацию. Отчет:

```

49 # оценить сеть
50 print("[ИИФ Оценка сети...") 51 предсказания=
model.predict(testX, batch_size=64) 52 print(classification_report(testY.argmax(axis=1),
53     прогнозы=argmax(ось=1), target_names=labelNames))

```

И с сохраением нашего графика потерь и точности на диск:

```

55 # график потерь и точности обучения 56 plt.style.use("ggplot")
57 plt.figure() 58 plt.plot(np.arange(0, 40), H.history["loss"], label=
"поеzd_потеря")

```

```

59 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
60 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
61 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
62 plt.title ("Тренировочные потери и точность на CIFAR-10")
63 plt.xlabel("Эпох а #")
64 plt.ylabel("Потери/точность")
65 plt.legend()
66 plt.savefig(аргументы["вывод"])

```

При оценке MiniVGGNet я провел два эксперимента:

1. Один с пакетной нормализацией.
2. Один без пакетной нормализации.

Давайте продолжим посмотреть эти результаты чтобы сравнить, как увеличивается производительность сети при применении пакетной нормализации.

15.3.1 С пакетной нормализацией

Чтобы обучить MiniVGGNet на наборе данных CIFAR-10, просто выполните следующую команду:

```
$ python minivggnet_cifar10.py --output output/cifar10_minivggnet_with_bn.png
[INFO] загрузка данных CIFAR-10...
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 50 000 образцов, проверка на 10 000 образцах
Эпох а 1/40
23 с - убыток: 1,6001 - акк: 0,4691 - val_loss: 1,3851 - val_acc: 0,5234
Эпох а 2/40
23 с - убыток: 1,1237 - акк: 0,6079 - val_loss: 1,1925 - val_acc: 0,6139
Эпох а 3/40
23s - убыток: 0,9680 - акк: 0,6610 - val_loss: 0,8761 - val_acc: 0,6909
...
Эпох а 40/40
23 с - убыток: 0,2557 - акк: 0,9087 - val_loss: 0,5634 - val_acc: 0,8236
[INFO] оценка сети...
      Точность      Вспомогательная поддержка f1-score
    самолет      0,88      0,81      0,85      1000
автомобиль      0,93      0,89      0,91      1000
    птица      0,83      0,68      0,75      1000
     Кот      0,69      0,65      0,67      1000
    олень      0,74      0,85      0,79      1000
                0,72      0,77      0,74      1000
    собака      0,85      0,89      0,87      1000
лягушка лошадь      0,85      0,87      0,86      1000
    корабль      0,89      0,91      0,90      1000
   грузовик      0,88      0,91      0,90      1000
среднее / общее      0,83      0,82      0,82      10000
```

На моем графическом процессоре эпохи были довольно быстрыми — 23 секунды на моем процессоре эпохи были значительно длиннее, x равняется 171 с.

После завершения обучения мы видим, что MiniVGGNet получает точность классификации 83%. На наборе данных CIFAR-10 с пакетной нормализацией — этот результат существенно выше, чем 60%

точности при применении ShallowNet в главе 12. Таким образом, мы видим, как более глубокая сетевая архитектура способна усвоить более богатые, более различительные признаки.

Но как насчет роли пакетной нормализации? Это действительно помогает нам здесь? Чтобы узнать, давайте перейти к следующему разделу.

15.3.2 Без пакетной нормализации

Вернитесь к реализации minivggnet.py и закомментируйте всю BatchNormalization.

слои, например:

```

27      # первый набор слоев CONV => RELU => CONV => RELU => POOL
28      model.add(Conv2D(32, (3, 3), padding="тоже самое",
29                      input_shape=inputShape))
30      model.add(Активация("relu"))
31      #model.add(Пакетная нормализация(ось=chanDim))
32      model.add(Conv2D(32, (3, 3), padding="тоже самое"))
33      model.add(Активация("relu"))
34      #model.add(Пакетная нормализация(ось=chanDim))
35      model.add(MaxPooling2D(pool_size=(2, 2)))
36      model.add(Выведение (0,25))

```

После того, как вы закомментировали все слои BatchNormalization из своей сети, повторно обучите MiniVGGNet на CIFAR-10:

```

$ python minivggnet_cifar10.py --output output/cifar10_minivggnet_without_bn.png
[INFO] загрузка данных CIFAR-10...
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 50 000 образцов, проверка на 10 000 образцах
Эпох а 1/40
13 с - убыток: 1,8055 - акк: 0,3426 - val_loss: 1,4872 - val_acc: 0,4573
Эпох а 2/40
13 с - убыток: 1,4133 - акк: 0,4872 - val_loss: 1,3246 - val_acc: 0,5224
Эпох а 3/40
13 с - убыток: 1,2162 - акк: 0,5628 - val_loss: 1,0807 - val_acc: 0,6139
...
Эпох а 40/40
13 с - убыток: 0,2780 - акк: 0,8996 - val_loss: 0,6466 - val_acc: 0,7955
[INFO] оценка сети...

```

точность	вспомнить поддержку f1-score
самолет	0,83
автомобиль	0,90
птица	0,75
Кот	0,64
олень	0,75
	0,69
собака	0,81
лягушка лошадь	0,85
корабль	0,90
грузовик	0,84
среднее / общее	0,79
	0,80
	0,79
	10000

Первое, что вы заметите, это то, что эта сеть обучается быстрее без пакетной нормализации (13 с по сравнению с 23 с, сокращение на 43%). Однако, как только сеть завершит обучение, вы заметите более низкую точность классификации — 79%.

Когда мыстроим MiniVGGNet с пакетной нормализацией (слева) и без пакетной нормализации (справа) бок о бок на рисунке 15.1, мыможем видеть положительное влияние пакетной нормализации на процесс обучения.

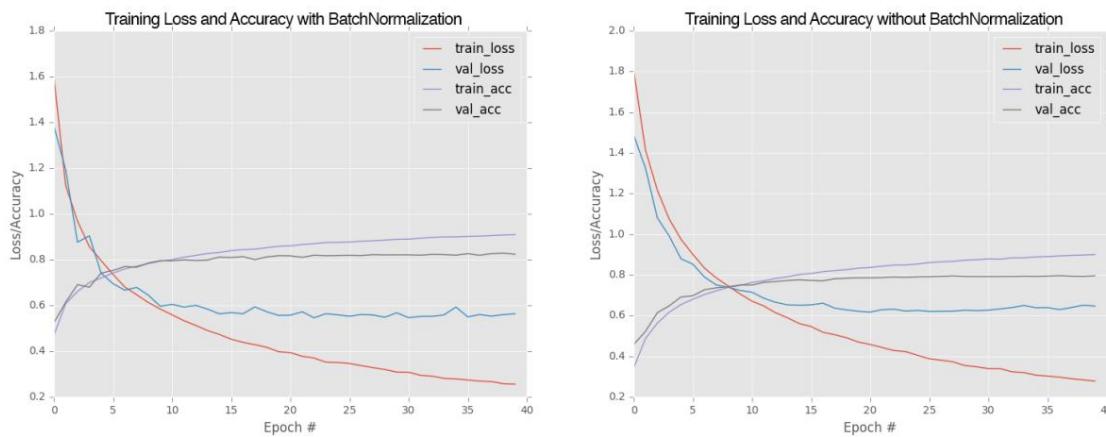


Рисунок 15.1: Слева: MiniVGGNet, обученная на CIFAR-10 с пакетной нормализацией. Справа: MiniVGGNet, обученный на CIFAR-10, без пакетной нормализации. Применение пакетной нормализации позволяет нам получить более высокую точность классификации и уменьшить влияние переобучения.

Обратите внимание, как потери для MiniVGGNet без пакетной нормализации начинают увеличиваться после 30-й эпохи, указывая на то, что сеть переоснащается обучающими данными. Мы также можем ясно видеть, что точность проверки стала достаточно насыщенной к эпохе 25.

С другой стороны реализация MiniVGGNet с пакетной нормализацией более стабильна. Хотя потери и точность начинают сглаживаться после эпохи 35, мы не так сильно переобучаемся — это одна из многих причин, почему мы предлагаем применять пакетную нормализацию к вашим собственным сетевым архитектурам.

15.4 Резюме

В этой главе мы обсудили семейство сверточных нейронных сетей VGG. CNN можно считать сетью VGG, если:

1. Он использует только фильтры 3×3 , независимо от глубины сети.
2. Существует несколько слоев $\text{CONV} \Rightarrow \text{RELU}$, применяемых перед одной операцией POOL , иногда с большим количеством слоев $\text{CONV} \Rightarrow \text{RELU}$, наложенных друг на друга по мере увеличения глубины сети.

Затем мы реализовали сеть, вдохновленную VGG, под одним названием MiniVGGNet. Эта сеть имела архитектуру состоявшую из двух наборов $(\text{CONV} \Rightarrow \text{RELU})^2 \Rightarrow$ слоев POOL , за которыми следовал набор слоев $\text{FC} \Rightarrow \text{RELU} \Rightarrow \text{FC} \Rightarrow \text{SOFTMAX}$. Мы также применяли пакетную нормализацию после каждой активации, а также отбраковку после каждого пула и полно связного слоя. Для оценки MiniVGGNet мы использовали набор данных CIFAR-10.

Наша предыдущая точность в CIFAR-10 составляла всего 60% от сети ShallowNet (глава 12). Однако с помощью MiniVGGNet мы смогли повысить точность до 83%.

Наконец, мы рассмотрели роль, которую пакетная нормализация играет в глубоком обучении и CNN.

нормализации, MiniVGGNet достиг точности классификации 83%, но без пакетной нормализации точность снизилась до 79% (и мы также начали видеть признаки переобучения).

Таким образом, ввод здесь таков: 1.

Пакетная нормализация может привести к более быстрой и стабильной сходимости с более высокой точностью.

2. Однако преимущества будут достигнуты за счет времени обучения — пакетная нормализация потребует больше «времени стены» для обучения сети, даже если сеть получит более высокую точность за меньшее количество эпох.

Тем не менее, дополнительное время обучения часто перевешивает негативные последствия и настоятельно рекомендуем вам применять пакетную нормализацию к вашей собственной сетевой архитектуре.

16. Планировщики скорости обучения

В нашей последней главе мы обучили архитектуру MiniVGGNet набору данных CIFAR-10. Чтобы смоделировать последствия переобучения, я представил концепцию добавления затухания к нашей скорости обучения при применении SGD для обучения сети.

В этой главе мы обсудим концепцию графиков скорости обучения, иногда называемую отжигом скорости обучения или аддитивной скоростью обучения. Регулируя скорость обучения от эпохи к эпохе, мы можем уменьшить потери, повысить точность и даже в определенных ситуациях сократить общее время, необходимое для обучения сети.

16.1 Снижение скорости обучения

Самые простые планировщики скорости обучения — это те, которые постепенно снижают скорость обучения со временем. Чтобы понять, почему графики скорости обучения являются интересным методом повышения точности модели, рассмотрим нашу стандартную формулу обновления веса из раздела 9.1.6:

`W += -args["альфа"] * градиент`

Напомним, что скорость обучения контролирует «шаг», который мы делаем по градиенту. Большие значения подразумевают, что мы делаем большие шаги, в то время как меньшие значения будут делать крохотные шаги — если равны нулю, сеть вообще не может делать никаких шагов (поскольку градиент, умноженный на ноль, равен нулю).

В наших предыдущих примерах в этой книге наши скорости обучения были постоянными — мы обычно не устанавливали $\alpha = \{0, 1, 0, 01\}$, а затем обучали сеть в течение фиксированного количества эпох без изменения скорости обучения. Этот метод может хорошо работать в некоторых ситуациях, но частично полезно снизить скорость обучения со временем.

При обучении нашей сети мы пытаемся найти какое-то место в нашем ландшафте потерь, где сеть получает разумную точность. Это не обязательно должно быть глобальным минимумом или даже локальным минимумом, но на практике достаточно просто найти область ландшафта потерь с достаточно низкими потерями.

Если мы будем постоянно поддерживать высокую скорость обучения, мы можем выйти за пределы этих областей с низкими потерями, поскольку мы будем делать слишком большие шаги, чтобы спуститься в эти области. Вместо этого мы можем уменьшить нашу скорость обучения тем самым позволяя нашей сети делать более мелкие шаги — эта уменьшенная скорость позволяет нашей сети спускаться в области ландшафта потерь, которые являются «более оптимальными» и в противном случае были бы полностью пропущены нашей системой. Большая скорость обучения

Таким образом, мы можем рассматривать процесс планирования скорости обучения

как: 1. Поиск набора достаточно «хороших» весов в начале процесса обучения с более высоким уровнем обучения показатель.

2. Настройка этих весов позже в процессе, чтобы найти более оптимальные веса, используя меньший скорость обучения

Существует два основных типа планировщиков скорости обучения с которыми вы, вероятно, столкнетесь: 1.

Планировщики скорости обучения, которые постепенно уменьшаются в зависимости от номера эпохи (например, линейный, полиномиальный или экспоненциальный функция).

2. Планировщики скорости обучения, которые снижаются в зависимости от конкретной эпохи (например, кусочная функция).

В этой главе мы рассмотрим оба типа планировщиков скорости обучения

16.1.1 Стандартный график распада в Keras

Библиотека Keras поставляется с планировщиком скорости обучения на основе времени — он управляет с помощью параметра затухания классов оптимизатора (например, SGD).

Возвращаясь к нашей предыдущей главе, давайте взглянем на блок кода, где мы инициализируем SGD и MiniVGGNet:

```
37 # инициализируем оптимизатор и
модель 38 print("[INFO] компилируем
модель...")
39 opt = SGD(lr=0,01, затухание=0,01 / 40, импульс=0,9,
нестеров=True) 40 model = MiniVGGNet.build(ширина = 32, высота = 32, глубина
= 3, классы = 10) 41 model.compile(loss="categorical_crossentropy", оптимизатор=opt,
метрики=["точность"])
42
```

Здесь мы инициализируем наш оптимизатор SGD со скоростью обучения = 0,01, импульсом = 0,9 и указываем, что мы используем склонный градиент Нестерова. Затем мы устанавливаем наше затухание (y) равным скорости обучения, деленным на общее количество эпох, для которых мы обучаем сеть (обычное эмпирическое правило), в результате чего получается $0,01/40 = 0,00025$.

Внутри Keras применяет следующий график скорости обучения, чтобы регулировать скорость обучения после каждой эпохи:

$$\alpha_{t+1} = \alpha_t \times \frac{1}{1 + y} \quad (16.1)$$

Если мы установим затухание равным нулю (значение по умолчанию в оптимизаторах Keras, если мы не укажем его явно), мы заметим, что это не влияет на скорость обучения (здесь мы произвольно устанавливаем нашу текущую эпоху равной $y = 1$, чтобы продемонстрировать этот точка):

$$\alpha_{t+1} = 0,01 \times 1 / (1 + 0,0 \times 1) = 0,01 \quad (16.2)$$

Но если мы место этого используем затухание = 0,01/40, вы заметите, что скорость обучения начнет уменьшаться после каждой эпохи (табл. 16.1).

Используя это затухание скорости обучения на основе времени, наша модель MiniVGGNet получила точность классификации 83%, как показано в главе 15. Я рекомендую вам установить затухание = 0 в оптимизаторе SGD.

Скорость обучения эпох и (α)	
1	0,01
2	0,00990
3	0,00971
...	...
38	0,00685
39	0,00678
40	0,00672

Таблица 16.1: Таблица, показывающая как наша скорость обучения уменьшается со временем, используя 40 эпох, начальная скорость обучения = 0,01 и срок затухания 0,04/40.

а затем повторите эксперимент. Взаместите, что есть также получает классификацию 83%.

точность; однако, поместив обучение графики двух моделей на рис. 16.1, в заместите, что что переобучение начинает происходить позже того, как потеря при проверке возрастают после 25-й эпохи (слева).

Этот результат отличается от того, когда мы установили затухание = 0,01 / 40 (справа) и получили гораздо более приятный результат. обучавшийся сюжет (и не говоря уже об более высокой точности). Используя снижение скорости обучения, мы часто можем только улучшить точность нашей классификации, но и уменьшить влияние переобучения тем самым увеличивая способность нашей модели обобщать.

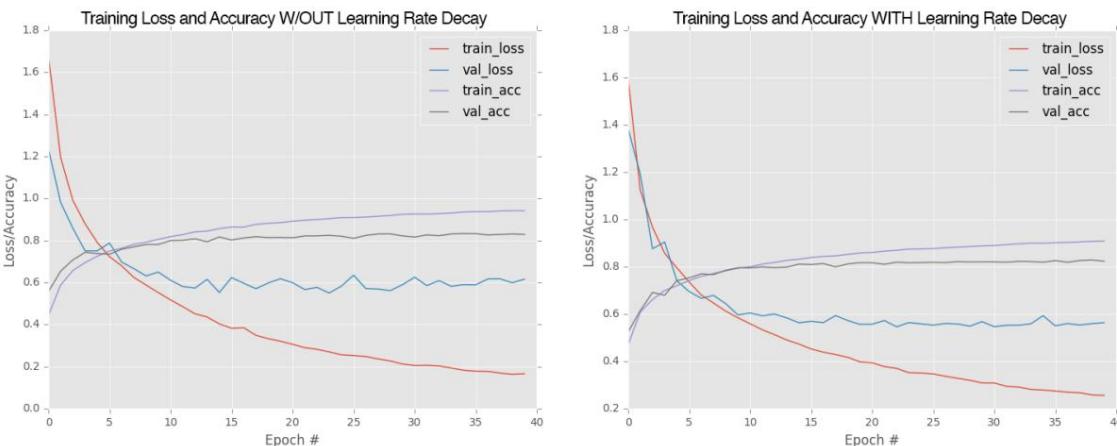


Рисунок 16.1: Слева: обучение MiniVGGNet на CIFAR-10 без снижения скорости обучения. Обратите внимание, как потеря начинает увеличиваться после эпохи 25, указывая на то, что происходит переобучение. Справа: применение коэффициента затухания 0,01/40. Это снижает скорость обучения со временем, помогая смягчить последствия переоснащения.

16.1.2 Ступенчатое затухание

Другой популярный планировщик скорости обучения — это пошаговое затухание, когда мы систематически отбрасываем скорость обучения после определенных эпох во время обучения. Планировщики скорости обучения затухания могут быть рассматриваться как кусочная функция, такая как на рис. 16.2. Здесь скорость обучения постоянна для числа эпох, затем падает и снова становится постоянным, затем снова падает и т. д.

При применении шагового затухания к нашей скорости обучения есть два варианта:

1. Определите уравнение, которое моделирует кусочное снижение скорости обучения, которое оно может достичь.

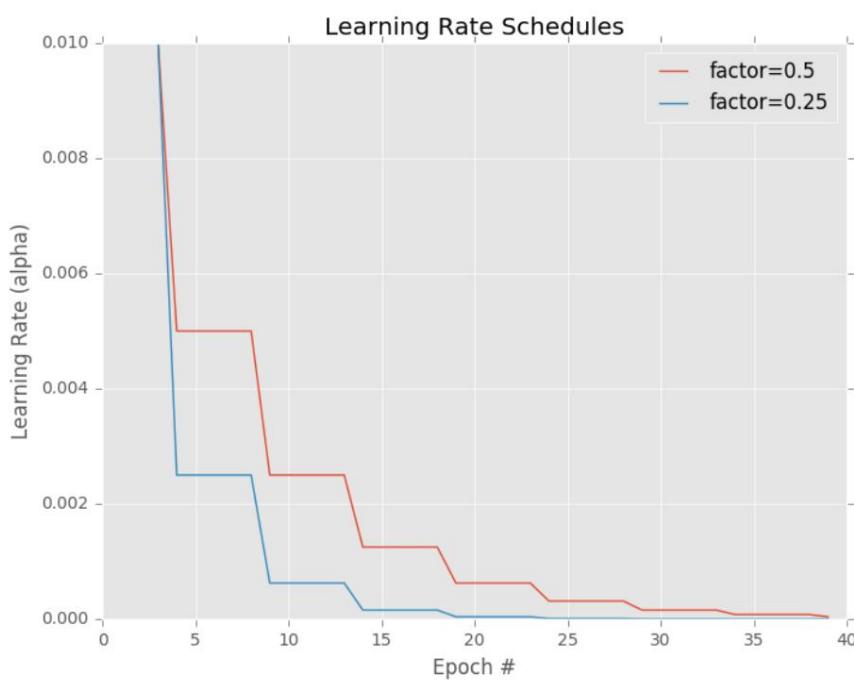


Рисунок 16.2: Пример двух графиков скорости обучения, которые снижают скорость обучения кусочно. Снижение значения фактора увеличивает скорость падения. В каждом случае скорость обучения приближается к нулю в последнюю эпоху.

2. Используйте то, что я называю методом Ctrl + C для обучения сети глубокого обучения где мы тренируемся течение некоторого количества эпох с заданной скоростью обучения в конечном итоге замечаем, что производительность проверки остановилась, затем Ctrl + C, чтобы остановить скрипт, скорректировать наше обучение оценивайте и продолжайте обучение.

В этой главе мы в первую очередь сосредоточимся на планировании скорости обучения на основе уравнения. Метод ctrl + c является более продвинутым и обычно применяется к большим наборам данных с использованием более глубоких нейронных сетей, где точное количество эпох, необходимо для получения разумной точности, неизвестно. Я подробно рассказываю об обучении ctrl + c в пакетах Practitioner Bundle и ImageNet Bundle этой книги.

При применении ступенчатого затухания мы часто снижаем скорость обучения либо(1) наполовину, либо(2) на порядок после каждого фиксированного числа эпох. Например, предположим, что наша начальная скорость обучения равна $\alpha = 0.1$. Через 10 эпох мы снижаем скорость обучения до $\alpha = 0.05$. Еще через 10 эпох обучения(т. е. 20-я общая эпох) α снова снижается до 0.025, так что $\alpha = 0.025$ и т. д. Фактически это точность такой же график скорости обучения изображенный на рисунке 16.2 выше (красная линия).. Синяя линия показывает гораздо более агрессивное падение с коэффициентом 0.25.

16.1.3 Внедрение пользовательских графиков скорости обучения в Keras

Удобно, что библиотека Keras предоставляет нам класс `LearningRateScheduler`, который позволяет нам определить пользовательскую функцию скорости обучения и затем автоматически применять ее в процессе обучения. Эта функция должна принимать номер эпохи в качестве аргумента, а затем вычислять желаемую скорость обучения на основе функции, которую мы определяем.

В этом примере мы определим кусочную функцию, которая снизит скорость обучения на

определенный фактор F через каждые D эпох . Таким образом, наше уравнение будет выглядеть так:

$$\alpha_E + 1 = \alpha_I \times F (1+E)/D \quad (16.3)$$

Где α_I — наша начальная скорость обучения F — значение фактора, контролирующего скорость, с которой скорость обучения падает, D — значение «отбрасывать каждые» эпохи, и E — текущая эпоха. Чем больше наш фактор F , тем медленнее будет снижаться скорость обучения И наоборот, чем меньше коэффициент F , тем быстрее будет снижаться скорость обучения

Написанное в коде Python, это уравнение может быть выражено как:

```
alpha = initAlpha * (фактор ** np.floor((1 + эпох a) / dropEvery))
```

Давайте продолжим реализуем этот пользовательский график скорости обучения, а затем применим его к MiniVG GNet на CIFAR-10. Откройте новый файл, назовите его cifar10_lr_decay.py и приступайте к написанию кода:

```
1 # установить бэкэнд matplotlib, чтобы оно можно было со временем работать в фоновом режиме 2
import matplotlib 3 matplotlib.use("Agg")  

  
4
5 # импортируем необходимые пакеты
из sklearn.preprocessing импортируем LabelBinarizer 7 из
sklearn.metrics ----- 12 импортировать
matplotlib.pyplot как plt 13 импортировать numpy как np 14
импортировать argparse
```

Строки 2-14 импортируют наши необходимые пакеты Python, как в оригинальном скрипте minivggnet_cifar10.py из главы 15. Однако обратите внимание на строку 9 , где мы импортируем наш LearningRateScheduler из библиотеки Keras — этот класс позволит нам определить нашу собственную скорость обучения планировщик.

Теперь давайте определим эту функцию:

```
16 по определению step_decay (эпох a):
17     # инициализировать базовую начальную скорость обучения коэффициент
18     отбрасывания # эпох и для отбрасывания каждого initAlpha = 0,01 factor = 0,25
19     dropEvery = 5
20
21
22
23     # вычислить скорость обучения для текущей эпохи
24     alpha = initAlpha * (factor ** np.floor((1 + epoch) / dropEvery))
25
26     # вернуть скорость обучения
27     return float(alpha)
```

Строка 16 определяет функцию step_decay , которая принимает единственный обязательный параметр — текущую эпоху . Затем мы определяем начальную скорость обучения(0,01), коэффициент отбрасывания(0,25), устанавливаем dropEvery = 5, подразумевая что мы будем снижать скорость обучения в 0,25 раза каждые пять эпох (строки 19-21).

Мы выбрали новую скорость обучения для текущей эпохи и в строке 24, используя значение 16.3 выше. Эта новая скорость обучения возврашается в вьювеющей функции в строке 27, что позволяет Keras внутренне обновлять скорость обучения оптимизатора.

Отсюда мы можем продолжить работу с нашим скриптом:

```

29 # построить разбор аргумента и разобрать аргументы
= argparse.ArgumentParser() 30 ap.add_argument("-o", "--output",
required=True,
31     help="путь к вью одному графику потерь/точности")
33 args = vars(ap.parse_args())
34
35 # загрузить данные обучения и тестирования, затем масшабировать их до
36 # диапазона [0, 1] 37 print("[INFO] загрузка данных CIFAR-10...")
38 ((trainX, trainY), (testX, testY)) = cifar10.load_data() 39 trainX =
trainX.astype("float") / 255.0 40 testX = testX.astype("float") / 255.0
41
42 # преобразовать метки из целых чисел в векторы 43 lb = LabelBinarizer() 44 trainY =
lb.fit_transform(trainY) 45 testY = lb.transform(testY)
46
47 # инициализировать имена меток для набора данных CIFAR-10
48 labelNames = ["самолет", "автомобиль", "птица", "кошка", "олень",
49     "собака", "лягушка", "лошадь", "корабль", "грузовик"]

```

Строки 30-33 анализируют наши аргументы командной строки. Нам нужен только один аргумент, --output, путь к нашему вью одному графику потерь/точности. Затем мы загружаем набор данных CIFAR-10 с диска и масшабируем интенсивность пикселей до диапазона [0,1] в строках 37-40. Строки 43-45 обрабатывают горячее кодирование меток классов.

Далее давайте обучим нашу сеть:

```

51 # определить набор обратных вьювов, которые будут переданы модели в время 52 # обучения 53 обратных вьювов
= [LearningRateScheduler(step_decay)]
54
55 # инициализируем оптимизатор и
модель 56 opt = SGD(lr=0.01, импульс=0.9, нестепров=True)
57 model = MiniVGGNet.build(width=32, height=32, depth=3, class=10) 58 model.
скомпилировать (потеря= "categorical_crossentropy", оптимизатор = опция
59     метрики=["точность"])
60
61 # обучаем сеть
62 H = model.fit(trainX, trainY, validation_data=(testX, testY),
63     batch_size=64, эпох и=40, callbacks=callbacks, verbose=1)

```

Строка 53 важна, так как она инициализирует наш список обратных вьювов. В зависимости от того, как определен обратный вьюв, Keras будет вызывать эту функцию в начале или в конце каждой эпохи, минипакетного обновления и т. д. LearningRateScheduler будет вызывать step_decay в конце каждой эпохи, позволяя нам обновлять обучение до начала, начало следующей эпохи и.

Строка 56 инициализирует оптимизатор SGD с импульсом 0,9 и ускоренным градиентом Нестрова. Параметр lr здесь будет проигнорирован, поскольку мы будем использовать обратный вьюв LearningRateScheduler, поэтому технически мы можем полностью исключить этот параметр; тем не менее, ях отел бывключить его чтобы он соответствовал initAlpha для ясности.

Строка 57 инициализирует MiniVGGNet, которую мы затем обучаем в течение 40 эпох на строках 62 и 63. Как только сеть обучена, мы можем оценить ее:

```
65 # оценить сеть
66 print("[ИНФ О оценка сети...") 67 прогнозы= model.predict(testX,
batch_size=64) 68 print(classification_report(testY.argmax(axis=1),
69     прогнозы.argmax(axis=1), target_names=labelNames))
```

А также график потерь и точности:

```
71 # график потерь и точности обучения72
plt.style.use("ggplot") 73 plt.figure() 74 plt.plot(np.arange(0,
40, H.history["loss"], label="train_loss") 75 plt.plot(np.arange(0,
40, H.history["val_loss"], label="val_loss") 76 plt.plot(np.arange(0, 40, H.history ["acc"], label="train_acc")
77 plt.plot(np.arange(0, 40, H.history["val_acc"], label="val_acc") 78 plt.title("Потери при обучении и
Точность по CIFAR-10") 79 plt.xlabel("Эпох а №") 80 plt.ylabel("Потери/точность") 81 plt.legend() 82
plt.savefig(args["output"])
```

Чтобы оценить влияние фактора отбрасывания на планирование скорости обучения общую точность классификации сети, мы будем оценивать два фактора отбрасывания 0,25 и 0,5 соответственно. Коэффициент снижения 0,25 будет уменьшаться значительно быстрее, чем показатель 0,5, как мы знаем из рисунка 16.2 выше.

Отдельно, обратите внимание, насколько быстрее фактор снижения 0,25 снижает нашу скорость обучения. Мы продолжим оценивать более быстрое падение скорости обучения на 0,25 (строка 20) — чтобы выполнить наш скрипт, просто введите следующую команду:

```
$ python cifar10_lr_decay.py --output output/lr_decay_f0.25_plot.png [INFO] загрузка данных CIFAR-10...
```

Обучение на 50000 выборках, проверка на 10000 выборках Эпох а
1/40 34 с - потеряя 1,6380 - акк: 0,4550 - val_loss: 1,1413 - val_acc: 0,5993
Эпох а 2/40 34 с - потеряя 1,1847 - акк: 0,5925 - val_loss: 1,0986 - val_acc: val_acc: 1,0986 0,6057

...

Эпох а 40/40 34
с - потеряя 0,5423 - акк: 0,8081 - val_loss: 0,5899 - val_acc: 0,7885 [INFO] оценка сети...

точность	вспомнить поддержку f1-score
самолет	0,81
автомобиль	0,91

самолет	0,81	0,81	0,81	1000
автомобиль	0,91	0,89	0,90	1000

птица	0,71	0,65	0,68	1000
Кот	0,63	0,60	0,62	1000
олень	0,72	0,79	0,75	1000
	0,70	0,67	0,68	1000
собака	0,80	0,88	0,84	1000
лягушка лошадь	0,86	0,83	0,84	1000
корабль	0,87	0,90	0,88	1000
грузовик	0,87	0,87	0,87	1000
среднее / общее	0,79	0,79	0,79	10000

Здесь мы видим, что наша сеть имеет точность классификации только 79%. Скорость обучения падает довольно агрессивно — после пятнадцатой эпохи и составляет всего 0,00125, что означает, что наша сеть делает очень маленькие шаги по ландшафту потерь. Это поведение можно увидеть на рисунке 16.3 (слева), где потеря достоверности и точность практически не изменились после пятнадцатой эпохи и.

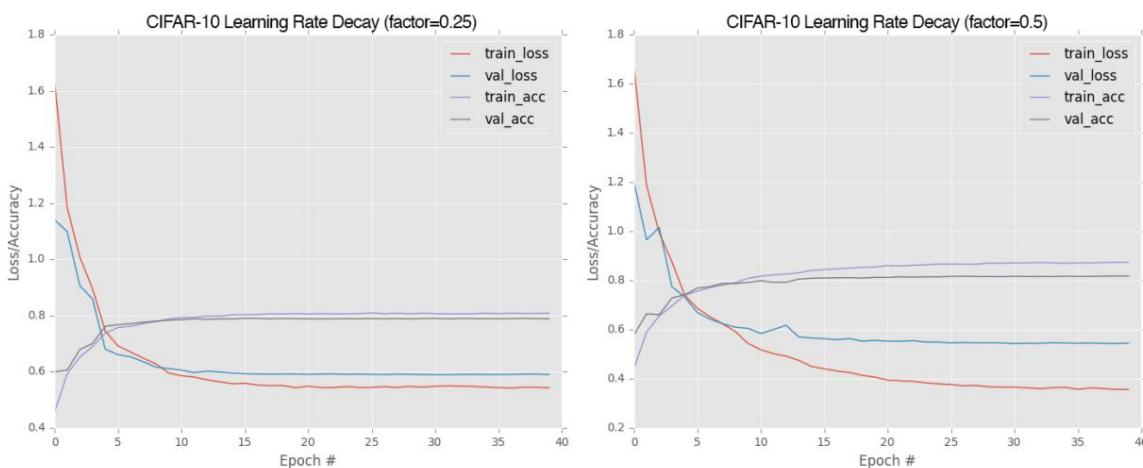


Рисунок 16.3: Слева: график точности/потери нашей сети с использованием более быстрого падения скорости обучения с коэффициентом 0,25. Обратите внимание, как потеря/точность стагнируют после эпох и 15, поскольку скорость обучения слишком мала. Справа: точность/потери нашей сети с более медленным падением скорости обучения (фактор = 0,5). Этот раз наша сеть может продолжать учиться после эпох и 30, пока не произойдет застой.

Если вместо этого мы изменим коэффициент снижения на 0,5, установив factor = 0,5 внутри step_decay:

16 по определению step_decay (эпох а):

```
17      # инициализировать базовую начальную скорость обучения коэффициент отбросвания
18      # эпохи и впадать каждые
19      инициализацияАльфа = 0,01
20      коэффициент = 0,5
21      drop каждый = 5
```

А затем повторите эксперимент, мы получим более высокую точность классификации:

```
$ python cifar10_lr_decay.py --output output/lr_decay_f0.5_plot.png
```

[INFO] загрузка данных CIFAR-10...

Обучение на 50 000 образцов, проверка на 10 000 образцах

```

Эпох а 1/40
35 с - убыток: 1,6733 - акк: 0,4402 - val_loss: 1,2024 - val_acc: 0,5771
Эпох а 2/40
34 с - убыток: 1,1868 - акк: 0,5898 - val_loss: 0,9651 - val_acc: 0,6643
...
Эпох а 40/40
33 с - убыток: 0,3562 - акк: 0,8742 - val_loss: 0,5452 - val_acc: 0,8177
[INFO] оценка сети...

```

	точность	вспомнить поддержку f1-score	
самолет	0,85	0,82	0,84
автомобиль	0,91	0,91	0,91
птица	0,75	0,70	0,73
Кот	0,68	0,65	0,66
олень	0,75	0,82	0,78
	0,74	0,74	0,74
собака	0,83	0,89	0,86
лягушка лягушка	0,88	0,86	0,87
корабль	0,89	0,91	0,90
грузовик	0,89	0,88	0,88
среднее / общее	0,82	0,82	0,82
			10000

На этот раз с более медленным падением скорости обучения мы получаем точность 82%. Глядя на сюжет на рис. 16.3 (справа) мы видим, что наша сеть продолжает обучаться в прошлые эпохи и 25-30 до потери застаеться на данных проверки. Прошлую эпоху а 30 скорость обучения очень мала на уровне $2.44e-06$ и не может внести какие-либо существенные изменения в веса, чтобы повлиять на потери/точность на данные проверки.

16.2 Резюме

Цель этой главы состояла в том, чтобы рассмотреть концепцию планировщиков скорости обучения и то, как они могут использоваться для повышения точности классификации. Мы обсудили два основных типа скорости обучения: планировщики:

1. Планировщики на основе времени, которые постепенно уменьшаются в зависимости от номера эпохи.
2. Планировщики на основе отбрасывания, которые отбрасываются в зависимости от определенной эпохи, аналогично поведению кусочной функции.

Какой именно планировщик скорости обучения вы должны использовать (если в вашем случае должны использовать планировщик) часть эксперимента. Как правило, в первом эксперименте не используется какой-либо тип планирования скорости затухания или обучения, чтобы вы могли получить базовую точность и кривую потерь/точности.

Оттуда вы можете ввести стандартное расписание, основанное на времени, предоставляемое Keras (с эмпирическое правило затухания = `alpha_init`/эпохи) и запустите второй эксперимент, чтобы оценить результаты. Следующие несколько экспериментов могут включать замену временного расписания на другое. Дорогоработанный с использованием различных драйверов.

В зависимости от того, насколько сложен ваш набор данных классификации, а также от глубины вашего сетей, вы можете выбрать метод Ctrl + C для обучения, как описано в практике.

Bundle и ImageNet Bundle — подход, используемый большинством специалистов по глубокому обучению, когда обучающие сети на наборе данных ImageNet.

В целом, будьте готовы потратить значительное количество времени на обучение своих сетей и оценку различных наборов параметров и процедур обучения. Даже простые наборы данных и проекты могут занять 10 секунд для получения экспериментов для получения моделей с высокой точностью.

На этом этапе изучения глубокого обучения вы должны понимать, что обучение глубоких нейронных сетей — это отчасти наука, отчасти искусство. Мягкая цель в этой книге — представить вам научные знания лежащие в основе обучения сети, а также общие эмпирические правила, которые я использую, чтобы вы могли изучить «искусство», стоящее за этим, но имейте в виду, что нечто сравнимое с реальным проведением экспериментов самостоятельно.

Чем больше у вас практики в обучении нейронных сетей, регистрации результатов того, что сделано, а что нет, тем лучше вы станете в этом. Когда дело доходит до освоения этого искусства, нет короткого пути — вам нужно потратить часы и освободиться от оптимизатором SGD (и другими) вместе с их параметрами.

17. Обнаружение недообучения и переобучения

Мы кратко коснулись недообучения и переобучения в главе 9. Теперь мы собираемся погрузиться глубже и обсудить как недообучение, так и переоснащение в контексте глубокого обучения. Чтобы помочь нам понять концепцию как недообучения так и переобучения я приведу нескольких графиков и рисунков, чтобы вы могли сопоставить с ними свои собственные кривые потерь/точности при обучении. машинное обучение/глубокое обучение, и вам раньше не приходилось замечать недостаточное/переоснащение.

Оттуда мы обсудим, как мы можем создать монитор обучения(пачти) в реальном времени для Keras, который вы можете использовать для наблюдения за процессом обучения вашей сети. До сих пор нам приходилось ждать, пока наша сеть завершит обучение, прежде чем мы можем построить график потерь и точности обучения.

Ожидание окончания процесса обучения для визуализации потерь и точности может оказаться расточительным с точки зрения времени, особенно если наши эксперименты занимают много времени и у нас нет возможности визуализировать потери/точность вовремя самого процесса обучения(кроме просмотра результатов). необработанный вывод терминала) — мы могли бы потратить часы или даже дни на обучение сети, не понимая, что процесс должен был быть остановлен после первых нескольких эпох .

Вместо этого было бы намного полезнее, если бы мы могли строить графики обучения и потерь после каждой эпохи и визуализировать результаты. Оттуда мы могли бы принимать более обоснованные решения относительно того, следует ли нам прекратить эксперимент раньше или продолжить обучение.

17.1 Чем такое недообучение и переоснащение?

При обучении собственных нейронных сетей вы должны быть очень обеспокоены как недообучением, так и переоснащением. Недообучение происходит, когда ваша модель не может получить достаточно низкие потери на тренировочном наборе. В этом случае ваша модель не может изучить основные закономерности в ваших обучающих данных . На другом конце спектра у нас есть переоснащение , когда ваша сеть слишком хорошо моделирует данные обучения и не может обобщить ваши данные проверки.

- Поэтому наша цель при обучении модели машинного обучения состоит в том,
- чтобы 1. Максимально уменьшить потери при обучении.
- 2. При обеспечении достаточно небольшого разрыва между потерями при обучении и тестировании.

Контролировать, может ли модель не соответствовать или переобучаться, можно путем настройки пропускной способности нейронной сети. Мы можем увеличить пропускную способность, добавив в нашу сеть больше слоев и нейронов. Точно так же мы можем уменьшить пропускную способность, удалив слои и нейроны, применив методы регуляризации (снижение веса, отсев, увеличение λ , ранняя остановка и т. д.).

Следующий рисунок 17.1 (вдохновленный прекрасным примером рисунка 5.3 Гудфеллоу и др., стр. 112 [10]) помогает визуализировать взаимосвязь между недообучением и переоснащением в сочетании с емкостью модели:

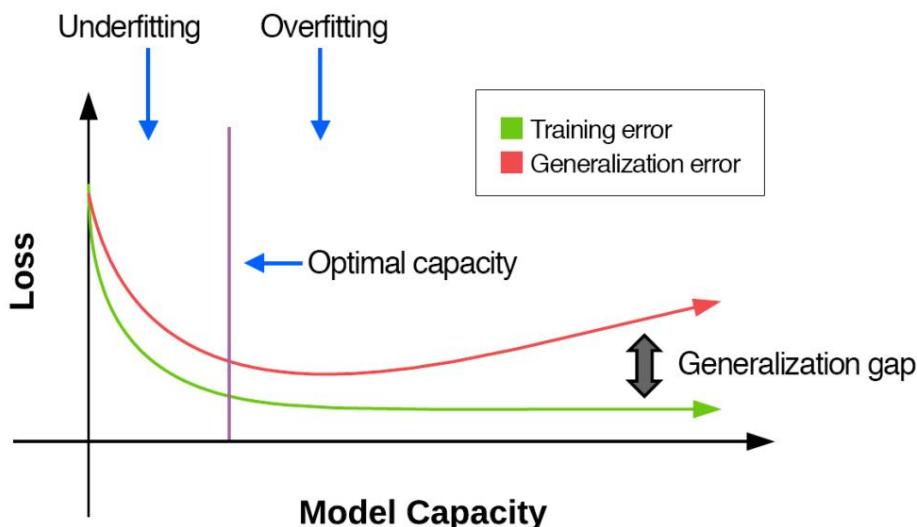


Рисунок 17.1: Связь между мощностью модели и потерями. Вертикальная фолетовая линия отделяет оптимальную емкость от недостаточной (слева) и чрезмерной (справа). Когда мы недообучаем, разрыв в обобщении со временем уменьшается. Оптимальная мощность возникает, когда потери как для обучения так и для обобщения сравниваются. Когда потери обобщения увеличиваются мы переоснащаемся. Примечание. Рисунок вдохновлен Goodfellow et al., стр. 112 [10].

По оси x мы отложили пропускную способность сети, а по оси y — потери, где более желательны более низкие потери. Однако, когда сеть начинает обучаться, мы находимся в «зоне недообучения» (рис. 17.1, слева). На данный момент мы просто пыгаемся изучить некоторые начальные шаблонные базовые данные и переместить веса от их случайных инициализаций к значениям, которые позволяют нам фактически «учиться» на самих данных. В идеале, как потери при обучении, так и потери при проверке будут падать вместе в течение этого периода — это падение демонстрирует, что наша сеть действительно обучается.

Однако по мере увеличения пропускной способности нашей модели (из-за более глубоких сетей, большего количества нейронов, отсутствия регуляризации и т. д.) мы достигнем «оптимальной пропускной способности» сети. В этот момент наши потери/точность обучения и проверки начинают расходиться друг с другом, и начинает формироваться заметный разрыв. Наша цель — ограничить этот разрыв, сохранив тем самым обобщаемость нашей модели.

Если нам не удастся ограничить этот разрыв, мы попадем в «зону переобучения» (рис. 17.1, справа). На этом этапе наши потери при обучении либо останутся на прежнем уровне, либо продолжат снижаться в той временной мере, как наши потери при проверке останутся на прежнем уровне и в конечном итоге возрастут. Увеличение потерь при проверке в течение ряда последовательных эпох является серьезным индикатором переобучения.

Итак, как бороться с переоснащением? В общем, есть два метода: 1.

Уменьшить сложность модели, выбрав более мелкую сеть с меньшим количеством слоев и

нейроны

2. Применять методы регуляризации.

Использование небольших нейронных сетей может работать для небольших наборов данных, но в целом это не предпочтительное решение. Вместо этого мы должны применять методы регуляризации, такие как снижение веса, отсев, увеличение данных и т. д. На практике почти всегда лучше использовать методы регуляризации для контроля переобучения, а не размера вашей сети [129], если только у вас нет веских оснований полагать, что ваша сеть является архитектурой просто слишком велика для этой проблемы.

17.1.1 Влияние скорости обучения на предыдущем

разделе мы рассмотрели пример переобучения и какую роль в этом играет наша скорость обучения? Есть ли на самом деле способ определить, является ли наша модель переоснащением с учетом набора гиперпараметров, просто изучив кривую потерь? Выдержите пару, что есть. Просто взгляните на рис. 17.2 в качестве примера (вдох новленный Карпаты и др. [93]).

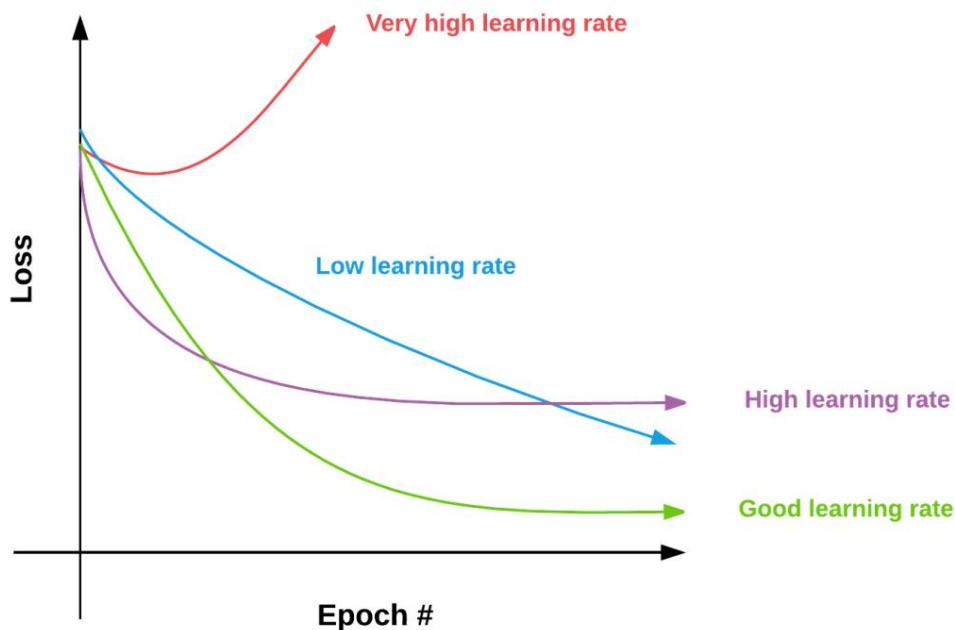


Рисунок 17.2: График, отображающий влияние различных скоростей обучения на потерю (в значительной степени вдох новлен Карпаты и др. [93]). Очень высокие скорости обучения (красные) сначала уменьшают потерю, но вскоре резко увеличиваются. Низкие скорости обучения (синие) будут примерно линейными по своим потерям с течением времени, в то время как высокие скорости обучения (фиолетовые) будут быстро падать, но возвращаться. Наконец, с меньшей экспоненциальной скоростью, что позволяет нам ориентироваться на ландшафте потерь.

По оси X мы отложили эпохи и нейронной сети вместе с соответствующими потерями по оси Y. В идеале наши потери при обучении и при проверке должны выглядеть как зеленая кривая, где потери быстро падают, но не настолько быстро, чтобы мы могли ориентироваться на ландшафте потерь и перейти в область с низкими потерями.

Кроме того, когда мы снова строим график потерь при обучении и проверке, мы можем получить еще более подробное представление о ходе обучения. Предпочтительно, чтобы наши потери при обучении и проверке почти имитировали друг друга, с небольшим разрывом между потерями при обучении и потерями при проверке, что указывает на небольшое переоснащение.

Однако в многих реальных приложениях идентичное имитирующее поведение просто нецелесообразно или даже нежелательно, поскольку может означать, что обучение нашей модели займет много времени. Поэтому нам просто нужно «учить» разрыв между обучением и потерей проверки. Пока разрыв не увеличивается резко, мы знаем, что существует приемлемый уровень переобучения. Однако, если нам не удается сократить этот разрыв, а потери при обучении и проверке сильно расщепляются, мы знаем, что рискуем переобучиться. Как только потери при проверке начинают увеличиваться, мы знаем, что сильно переоснащаемся.

17.1.2 Обратите внимание на свои тренировочные кривые

При обучении собственных нейронных сетей обратите внимание на кривые потерь и точности как для данных обучения так и для проверки. В течение первых нескольких эпох может показаться, что нейронная сеть хорошо отслеживается возможно, но немного недообучается, но эта картина может быстро измениться и вы можете начать видеть расщепление в обучении и потерях при проверке. Когда это произойдет, оцените свою модель:

- Применяете ли вы какие-либо методы регуляризации? •
- Ваша скорость обучения слишком высока? • Ваша сеть слишком глубока?

Отдельно, обучение сетей глубокого обучения — это отчасти наука, отчасти искусство. Лучший способ научиться читать эти кривые — обучить как можно больше сетей и изучить их графики. Современное выразите понимание того, что работает, а что нет, но не ждите, что у вас все получится «правильно» с первой попытки. Даже самый опытный специалист по глубокому обучению проведет от 10 до 100 экспериментов, погружаясь в кривые потери/точности, отмечая что работает, а что нет, и, в конце концов, найдет решение, которое работает.

Наконец, вы также должны принять тот факт, что переобучение для определенных наборов данных неизбежно. Например, очень легко подогнать модель под набор данных CIFAR-10. Если ваши потери при обучении и при проверке начинают расщепляться не паникуйте — просто попытайтесь максимально контролировать разрыв.

Также имейте в виду, что помимо того, как вы снижаете скорость обучения в более поздние эпохи (например, при использовании планировщика скорости обучения), вам также будет легче переобучиться. Этот момент станет более ясным в более продвинутых главах комплекта Practitioner Bundle и ImageNet Bundle.

17.1.3 Что делать, если потери при валидации ниже потерь при обучении?

Еще одно странное явление, с которым вы можете столкнуться — это когда ваши потери при валидации на самом деле ниже, чем ваши потери при обучении. Как это возможно? Как сеть может лучше работать с проверочными данными, когда шаблоны, которые она пытается изучить, взяты из обучающих данных?

Разве эффективность обучения всегда должна быть лучше, чем потеря при проверке или тестировании?

Не всегда. На самом деле причин такого поведения несколько. Возможно, самое простое объяснение:

1. В ваших обучающих данных видны «сложные» примеры для классификации.
2. В то время как ваши данные проверки состоят из «легких» точек данных.

Однако, если только вы не отобрали свои данные таким образом, мало вероятно, что случайное разделение обучения и тестирования аккуратно сегментирует эти типы точек данных.

Второй причиной может быть увеличение данных. Мы подробно рассматриваем увеличение данных в пакете Practitioner Bundle, но суть в том, что в процессе обучения мы случайным образом изменяем обучающие изображения, применяв к ним случайные преобразования, такие как перемещение, вращение, изменение размера и сдвиг. Из-за этих изменений сеть постоянно видит дополнительные примеры обучающих данных, чтобы извлечь форму регуляризации, позволяющей сети лучше обобщать проверочные данные, но, возможно, работать уже на обучающем наборе (подробнее об этом см. в главе 9.4). регуляризация).

Третьей причиной может быть то, что вы недостаточно тренируетесь. Возможно, в этом случае рассмотреть увеличение скорости обучения и настройка силы регуляризации.

17.2 Мониторинг тренировочного процесса

В первой части этого раздела мы создадим обратный вьюв TrainingMonitor, который будет возвращаться в конец каждой эпохи и при обучении сети с помощью Keras. Этот монитор будет сериализовать потери и точность как для обучения, так и для проверки, установленная на диск, с последующим построением графика данных.

Применение этого обратного вьюва вовремя обучения позволит нам следить за процессом обучения и определять раннее переобучение, что позволяет нам прервать эксперимент и продолжить попытки настроить наши параметры.

17.2.1 Создание тренировочного монитора

Наш класс TrainingMonitor будет жить в подмодуле pyimagesearch.callbacks:

```
|--- pyimagesearch
| |--- __init__.py
| |--- обратные вьювы
| | |--- __init__.py
| | |--- trainingmonitor.py
| |--- наборы данных
| |--- нн
| |--- предварительная обработка
| |--- утилиты
```

Создайте файл trainingmonitor.py и приступим:

```
1 # импортируем необязательные пакеты
2 из keras.callbacks импортировать BaseLogger
3 импортировать matplotlib.pyplot как plt
4 импортировать numpy как np
5 импортировать json
6 импорт OS
7
8 класс TrainingMonitor(BaseLogger):
9     def __init__(я figPath,(jsonPath=False, startAt=0):
10         # со временем вьюк одной путь для фигуры путь к JSON
11         # сериализованной файл и начальная эпоха
12         супер(TrainingMonitor, сам).__init__()
13         self.figPath = figPath
14         self.jsonPath = jsonPath
15         self.startAt = startAt
```

Строки 1-6 импортируют необязательные пакеты Python. Чтобы создать класс, который регистрирует наши потери и точность на диске, нам нужно расширить класс Keras BaseLogger (строка 2).

Конструктор класса TrainingMonitor определен в строке 9. Конструктор требует

один аргумент, за которым следуют два необязательных:

- figPath: путь к вьюку одному графику, который мы можем использовать для визуализации потерь и точности во времени
- jsonPath: необязательный путь, используемый для сериализации значений потерь и точности в виде файла JSON. Этот путь полезен, если вы хотите использовать историю обучения для создания собственных графиков.
- startAt: это начальная эпоха, в которой возобновляется обучение при использовании обучения ctrl + с. Мы рассматриваем обучение ctrl + с в пакете Practitioner Bundle, поэтому мы можем игнорировать эту переменную для теперь.

Затем давайте определим обратный вьюв on_train_begin, который, как следует из названия, называется один раз при запуске тренировочного процесса:

```

17     def on_train_begin(я журналы= {}):
18         # инициализируем словарь истории
19         сам.Ч = {}
20
21         # если путь истории JSON существует, загрузить историю обучения
22         если self.jsonPath не None:
23             если os.path.exists(self.jsonPath):
24                 self.H = json.loads(open(self.jsonPath).read())
25
26             # проверить, была ли указана начальная эпох
27             если self.startAt > 0:
28                 # цикл по записям в журнале истории и
29                 # обрезать все записи, которые нах одятся за начальным
30                 # эпох а
31                 для k в self.H.keys():
32                     self.H[k] = self.H[k][:self.startAt]

```

В строке 19 мы определяем `H`, который используется для представления «истории» убытков. Посмотрим, как этот словарь обновляется в функции `on_epoch_end` в следующем блоке кода.

Строка 22 проверяет, был ли указан путь JSON. Если это так, мы затем проверяем, чтобы видеть, если этот JSON-файл существует. При условии, что файл JSON существует, мы загружаем его содержимое и обновляем историю словарь `H` вплоть до начальной эпохи (поскольку оттуда мы продолжим обучение).

Теперь мы можем перейти к самой важной функции, `on_epoch_end`, которая вызывается когда эпоха обучения завершается:

```

34     def on_epoch_end(я эпох а, журналы= {}):
35         # перебираем логи и обновляем потери, точность и т.д.
36         # на весь тренировочный процесс
37         для(k, v) в logs.items():
38             l = self.H.get(k, [])
39             л. добавить(v)
40             self.H[k] = l

```

Метод `on_epoch_end` автоматически передается параметрам из Keras. Первый целое число, представляющее номер эпохи. Второй — это словарь `logs`, который содержит потери при обучении и проверке + точность для текущей эпохи. Мы перебираем каждый из элементов в `logs`, а затем обновить наш словарь истории (строки 37-40).

После выполнения этого кода словарь `H` теперь имеет четыре ключа:

1. поезд_потеря
2. поезд_акк
3. val_loss
4. val_acc

Мы поддерживаем список значений для каждого из этих ключей. Каждый список обновляется конце каждого эпохи, что позволяет нам построить обновленную кривую потерь и точности, как только эпоха завершится.

В случае, если был предоставлен `jsonPath`, мы сериализуем историю `H` на диск:

```

42         # проверяем, следует ли сериализовать историю обучения
43         # в файл
44         если self.jsonPath не None:
45             f = открыть(self.jsonPath, "w")

```

```
46     f.write(json.dumps(self.H))
47     f.close()
```

Наконец, мы также можем построить реальный сюжет:

```
49         # убедитесь, что прошло как минимум две эпохи перед построением графика
50         # (эпохи начинаются с нуля)
51         если len(self.H["потеря"]) > 1:
52             # построить график потери и точности обучения
53             N = np.arange(0, len(self.H["потеря"]))
54             plt.style.use("ggplot")
55             plt.figure()
56             plt.plot(N, self.H["потеря"], метка="train_loss")
57             plt.plot(N, self.H["val_loss"], label="val_loss")
58             plt.plot(N, self.H["acc"], label="train_acc")
59             plt.plot(N, self.H["val_acc"], метка="val_acc")
60             plt.title("Потери при обучении и точность [эпохи {}].format(
61                 len(self.H["потеря"])))
62             plt.xlabel("Эпохи #")
63             plt.ylabel("Потери/точность")
64             plt.legend()
65
66             # сохранить фигуру
67             plt.savefig(self.figPath)
68             plt.close()
```

Теперь, когда наш TrainingMonitor определен, давайте перейдем непосредственно к мониторингу и присмотримся к нему.

17.2.2 Обучение присмотру за деталями

Для мониторинга процесса обучения нам потребуется создать скрипт драйвера, который обучает сеть с помощью обратного вьювов TrainingMonitor. Для начала откройте новый файл, назовите его cifar10_monitor.py и вставьте следующий код:

```
1 # установить бэкенд matplotlib, чтобы фигуры можно было сохранять в формате png
2 импортировать matplotlib
3 matplotlib.use("Agg")
4
5 # импортируем необходимые пакеты
6 из pyimagesearch.callbacks импорта TrainingMonitor
7 из sklearn.preprocessing импорта LabelBinarizer
8 из pyimagesearch.nn.conv импортировать MiniVGGNet
9 из keras.optimizers импорт SGD
10 из keras.datasets импортировать cifar10
11 импортировать синтаксический анализ
12 импорт OS
```

Строки 1-12 импортируют необходимые пакеты Python. Обратите внимание, как мы импортируем наши недавно определенные классы TrainingMonitor, чтобы мы могли следить за обучением нашей сети.

Далее мы можем проанализировать наши аргументы командной строки:

```

14 # построить аргумент parse и разобрать аргументы15 ap =
argparse.ArgumentParser() 16 ap.add_argument("-o", "--output",
required=True, help="путь к въю одному каталогу") 18 args = варс
17     (ap.parse_args())

19
20 # показать инф ормацию об ID процесса 21
print("[INFO ID процесса: {}".format(os.getpid()))

```

Единственный аргумент командной строки, который нам нужен это --outrit, путь к въю одному каталогу для сохраните нашу сгенерированную цифру matplotlib и сериализованную историю обученияJSON.

Мне нравится использовать изящный трюк — использовать идентификатор процесса, присвоенный операционной системой, дляименованияграфиков и файлов JSON. Если язамечаю, что обучение идет плохо, я могу просто открыть свой диспетчер задач и убить идентификатор процесса, связанный с моим скриптом. Эта возможность особенно полезна, если вы проводите несколькоэкспериментов одновременно. Стока 21 просто отображает идентификатор процесса на нашемэкране.

Оттуда мы въполняем нашстандартный конвейер загрузки набора данных CIFAR-10 и подготовки данные + метки дляобучения

```

23 # загрузить данные обучения и тестирования затем масшабировать их
в диапазоне 24 # [0, 1] 25 print("[INFO] загрузка данных CIFAR-10...") 26 ((trainX,
trainY), (testX, testY)) = cifar10.load_data() 27 trainX = trainX.astype("float") / 255,0 28
testX = testX.astype("float") / 255,0

```

```

29
30 # преобразовать метки из цельк чисел в векторы31
lb = LabelBinarizer() 32 trainY = lb.fit_transform(trainY) 33 testY =
lb.transform(testY)

```

```

34
35 # инициализировать имена меток для набора данных CIFAR-10
36 labelNames = ["самолет", "автомобиль", "птица", "кошка", "олень", "собака",
37     "лягушка", "лошадь", "корабль", "грузовик"]

```

Теперь мы готовыинициализировать оптимизатор SGD вместе с архитектурой MiniVGGNet:

```

39 # инициализируем оптимизатор SGD , но без снижения скорости обучения(ширина
= 32, въсота = 32, глубина = 3, классы= 10) 43 model.compile(loss="categorical_crossentropy",
оптимизатор=opt,

```

```

44     метрики=["точность"])

```

Обратите внимание, чтояне включают снижение скорости обучения. Этоупущение намеренно, поэтому я могу продемонстрировать, как конгролировать ваштренировочный процесс и въявлять переобучение помере его возникновения

Давайте создадимнаш обратный възов TrainingMonitor и обучимсеть:

```

46 # построить набор обратных възовов
47 figPath = os.path.sep.join([args["output"], "{}.png".format( os.getpid())])
48

```

```

49 jsonPath = os.path.sep.join([args["output"], "{}.json".format(
50         os.getpid())]) 51
обратный в ввод = [TrainingMonitor(figPath, jsonPath=jsonPath)]
52
53 # обучаем сеть
54 print("И НФ О обучают шесть...") 55 model.fit(trainX, trainY,
validation_data=(testX, testY),
56     batch_size=64, эпох и=100, callbacks=callbacks, verbose=1)

```

Строки 47-50 инициализируют пути к нашему вью одному графику и сериализованному файлу JSON соответственно. Обратите внимание, что каждый из этих путей к файлам включает идентификатор процесса, что позволяет нам легко связать эксперимент с идентификатором процесса — в случае, если эксперимент пойдет плохо, мы можем отключить сценарий с помощью нашего диспетчера задач. Учитывая фигуру и пути JSON, строка 51 строит наш список обратных вводов, состоящий из одной записи, самого TrainingMonitor.

Наконец, линии 55 и 56 обучают нашу сеть в общей сложности 100 эпох. Я намеренно установил очень высокие эпохи, чтобы побудить нашу сеть к переобучению.

Чтобы выполнить скрипт (и научиться обнаруживать переоснащение), просто выполните следующую команду:

```
$ python cifar10_monitor.py --вью одной ввод
```

После первых нескольких эпох вы заметите два файла в вашем--output каталоге:

```
$ ls вью од/
7857.json 7857.png
```

Это ваша сериализованная история тренировок и графики обучения соответственно. Каждый файл назван в честь идентификатора процесса, создавшего их. Преимущество использования TrainingMonitor заключается в том, что теперь мы можем наблюдать за процессом обучения и контролировать его после завершения каждой эпохи.

Например, на рис. 17.3 (вверху слева) показан график потерь и точности после эпох 5 и 5. Прямо сейчас мы все еще находимся в «зоне недообучения». Наша сеть явно обучается, обучаясь данным, поскольку мы видим, что потери уменьшаются а точность увеличивается одновременно достигла плато.

После эпох 10 мы можем заметить признаки переоснащения, но ничего, чтобы выявляло чрезмерную тревогу (рис. 17.3, вверху посередине). Потери при обучении начинают отличаться от потерь при проверке, но некоторое расхождение совершенно нормально и даже является оральным признаком того, что наша сеть продолжает изучать основные закономерности из обучаемых данных.

Однако к эпохе 25 мы достигли «зоны переобучения» (рис. 17.3, вверху справа). Потери при обучении продолжают снижаться в то время как потери при проверке остаются на прежнем уровне. Это явный первый признак переоснащения и грядущих плохих вещей.

К эпохе 50 у нас явно проблемы (рис. 17.3, внизу слева). Потери при валидации начинают увеличиваться, что является ярым признаком переобучения. На этом этапе было бы лучше остановить эксперимент, чтобы переоценить свои параметры.

Если бы мы позвонили сети обучаться до эпохи 100, переобучение только удалилось бы (рис. 17.3, внизу справа). Разрыв между потерями при обучении и потерями при валидации огромен, в то время как потери при валидации продолжают расти. Хотя точность проверки этой сети превышает 80%, способность этой модели к обобщению будет довольно низкой. Основываясь на этих графиках, мы можем ясно видеть, когда и где начинает происходить переоснащение. При проведении собственных экспериментов убедитесь, что вы используете TrainingMonitor, чтобы контролировать процесс обучения.

Наконец, когда вы начинаете думать, что есть признаки переобучения, не становитесь слишком счастливыми, чтобы забыть эксперимент. Позвольте сети обучаться еще 10-15 эпох, чтобы убедиться, что ваша догадка верна.

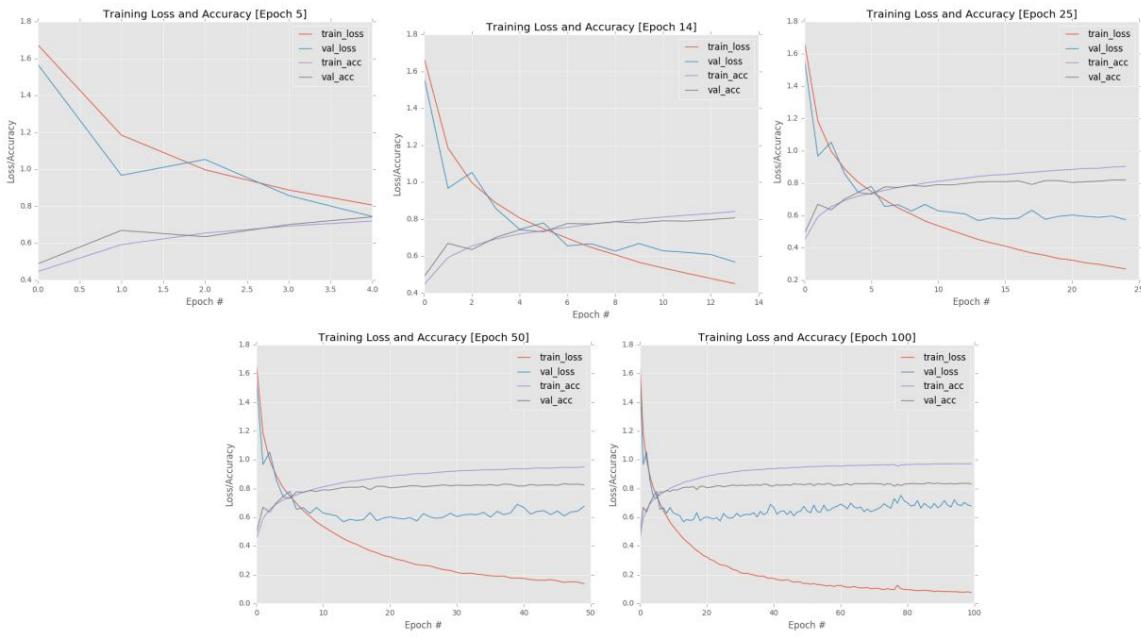


Рисунок 17.3: Примеры мониторинга процесса обучения путем изучения кривых потерь при обучении/валидации. В эпоху 5 мы видим еще нах одимся в зоне недообучения. В эпоху 14 мы начинаем переобучаться, но не очень сильно беспокоиться. К эпохе 25 мы определенно нах одимся нутри зоны переобучения. Эпохи 50 и 100 демонстрируют сильное переоснащение, поскольку потери при проверке растут, а потери при обучении продолжают падать.

правильно, и что происходит — нам часто нужен контекст этих эпох, чтобы помочь нам принять это окончательное решение.

Слишком часто я вижу, как специалисты по глубокому обучению, плохо знакомые с машинным обучением, слишком рано запускают эксперименты, убивающие их. Подождите, пока не увидите явных признаков переобучения, затем завершите процесс. Оттачивая свои навыки глубокого обучения, вы разовьете чувство, которое будет направлять вас при обучении ваших сетей, и дотех пор доверяйте контексту дополнительных эпох, чтобы вы могли принимать болеезвезденные и обоснованные решения.

17.3 Резюме

В этой главе мы рассмотрели недообучение и переоснащение. Недообучение происходит, когда ваша модель не может получить достаточно низкие потери на тренировочном наборе. Между тем, переобучение происходит, когда разрыв между вашими потерями при обучении и потерями при проверке слишком велик, что указывает на то, что сеть слишком сильно моделирует базовые шаблоны обучая щих данных.

С недообучением относительно легко бороться просто добавьте в свою сеть больше слоев/нейронов. Однако переоснащение — это совсем другой зверь. При переоснащении следует учитывать:

1. Уменьшение пропускной способности вашей сети путем удаления слоев/нейронов (не рекомендуется если только для наибольшего набора данных).
2. Применение более сильных методов регуляризации.

Почти в всех ситуациях вы должны начать попытаться применить более сильную регуляризацию, чем уменьшать размер вашей сети, за исключением случаев, когда вы пытааетесь обучить очень глубокую сеть на крошечном наборе данных.

Поняв взаимосвязь между емкостью модели и недообучением и переоснащением, мы научились отслеживать процесс обучения и выявлять переоснащение по мере его возникновения.

процесс позволяет нам установить обучение сетей на раннем этапе вместо того, чтобы тратить время на то, чтобы позволить сети переоснащение. Наконец, мы завершили главу, рассмотрев несколько красноречивых примеров переобучения

18. Модели контрольных точек

В главе 13 мы обсуждали, как сохранять и сериализовать ваши модели на диск после завершения обучения. А в последней главе мы узнали, как вводить недообучение и переоснащение по мере их возникновения, что позволяет вам отсеивать эксперименты, которые не работают, со временем при этом перспективные модели в время обучения.

Однако вам может быть интересно, можно ли комбинировать обе эти стратегии. Можно ли мы сериализовать модели в каждый раз, когда наши потери/точность улучшаются? Или можно сериализовать только лучшую модель (т. е. модель с наименьшими потерями или с наибольшей точностью) в процессе обучения? Выдержите пари. И, к счастью, нам также не нужно создавать собственный обратный ввод — эта функциональность встроена прямо в Keras.

18.1 Контрольные точки Улучшения модели нейронной сети

Хорошим применением контрольных точек является сериализация вашей сети на диск каждый раз, когда в время обучения происходит улучшение. Мы определяем «улучшение» как уменьшение потерь или повышение точности — мы установим этот параметр внутри фактического обратного ввода Keras.

В этом примере мы будем обучать архитектуру MiniVGGNet на наборе данных CIFAR-10, а затем сериализовать наши веса сети на диск каждый раз, когда производительность модели улучшается. Для начала откройте новый файл, назовите его `cifar10_checkpoint_improvements.py` и вставьте следующий код:

```
1 # импортируем необходимые пакеты2
из sklearn.preprocessing import LabelBinarizer 3 из
pyimagesearch.nn.conv import MiniVGGNet 4 из keras.callbacks
import ModelCheckpoint 5 из keras.optimizers import SGD 6 из
keras.datasets import cifar10 7 из argparse 8 из
os
```

Строки 2-8 импортируют необъодимые пакеты Python. Обратите внимание на класс ModelCheckpoint, импортированный в строку 4 — этот класс позволяет нам создавать контрольные точки и сериализовать наши сети на диск в каждый раз, когда мы обнаруживаем постепенное улучшение производительности модели.

Далее, давайте проанализируем наши аргументы командной строки:

```
10 # построить разбор аргумента и разобрать аргументы 11 ap
= argparse.ArgumentParser() 12 ap.add_argument("-w", "-weights",
required=True,
13           help="путь к каталогу весов") 14 args
= vars(ap.parse_args())
```

Единственный аргумент командной строки, который нам нужен, — это --weights, путь к вьюк одному каталогу, в котором будут храниться наши сериализованные модели в процессе обучения. Затем мы выполняем нашу стандартную процедуру загрузки набора данных CIFAR-10 с диска, масштабируем интенсивность пикселей до диапазона [0,1], а затем выполняем горячее кодирование меток:

```
16 # загрузить данные обучения и тестирования, затем масштабировать их до 17 # диапазона [0, 1] 18 print("[INFO] загрузка данных CIFAR-10...")
19 ((trainX, trainY), (testX, testY)) = cifar10.load_data() 20 trainX =
trainX.astype("float") / 255.0 21 testX = testX.astype("float") / 255.0
```

```
22
23 # преобразовать метки из целых чисел в векторы 24
lb = LabelBinarizer() 25 trainY = lb.fit_transform(trainY) 26 testY =
lb.transform(testY)
```

Учитывая наши данные, мы теперь готовы инициализировать наш оптимизатор SGD вместе с архитектурой MiniVGGNet:

```
28 # инициализируем оптимизатор и
модель 29 print("[INFO] компилируем
модель...") 30 opt = SGD(lr=0.01, затухание=0.01 / 40, импульс=0.9,
истеров=True) 31 model = MiniVGGNet.build(ширина = 32, высота = 32, глубина
= 3, классы = 10) 32 model.compile(loss="categorical_crossentropy", оптимизатор=opt,
33               метрики=["точность"])
```

Мы будем использовать оптимизатор SGD с начальной скоростью обучения = 0,01, а затем медленно уменьшать ее в течение 40 эпох. Мы также применим импульс $\gamma = 0.9$ и укажем, что также следует использовать ускорение Нестерова.

Архитектура MiniVGGNet создается для приема вх однок изображений шириной 32 пикселя в высотой 32 пикселями глубиной 3 (количество каналов). Мы устанавливаем class=10, так как набор данных CIFAR-10 имеет десять возможных меток классов.

Критический шаг для проверки нашей сети можно найти в блоке кода ниже:

```
35 # создать обратный вьюв для сохранения на диск только *лучшей* модели 36
# на основе потери проверки
37 fname = os.path.sep.join([аргумент["веса"],
                           "веса-{Эпох а:03d}-{val_loss:.4f}.hdf5"])
38
```

```
39 контрольная точка = ModelCheckpoint(fname, monitor="val_loss", mode="min",
40         save_best_only=True, verbose=1)
41 обратный вивов = [контрольная точка]
```

В строках 37 и 38 мы создаем специальную строку шаблона имени файла (fname), которую Keras использует при записи наших моделей на диск. Первая переменная в шаблоне {epoch:03d} — это номер нашей эпохи, записанный тремя цифрами.

Вторая переменная — это метрика, которую мы хотим отслеживать для улучшения {val_loss:4f}, сама потеря для проверки, установленная текущую эпоху. Конечно, если мы хотим следить за точностью проверки, мы можем заменить val_loss на val_acc. Если бы мы в месте этого отели отслеживать потери и точность обучения переменная стала бы train_loss и train_acc соответственно (хотя бы рекомендовал отслеживать ваши показатели проверки, поскольку они дадут вам лучшее представление о том, как ваша модель будет обобщаться).

Как только шаблон в строке одного имени файла определен, мы создаем экземпляр класса ModelCheckpoint в строках 39 и 40. Первый параметр ModelCheckpoint — это строка, представляющая наш шаблон имени файла. Затем мы передаем то, что мы хотели бы контролировать. В этом случае мы хотим отслеживать потерю проверки (val_loss).

Параметр режима определяет, должны ли ModelCheckpoint искать значения, которые минимизируют нашу метрику или максимизируют ее. Так как мы работаем с потерями, чем меньше, тем лучше, поэтому мы устанавливаем mode="min". Если бы вместо этого мы работали с val_acc, мы бы установили mode="max" (поскольку чем выше точность, тем лучше).

Установка save_best_only=True гарантирует, что последняя лучшая модель (в соответствии с отслеживаемой метрикой) не будет перезаписана. Наконец, параметр verbose=1 просто регистрирует уведомление на нашем терминале, когда модель сериализуется на диск в время обучения.

Затем в строке 41 создается список обратных вивов — единственный обратный вивов, который нам нужен — это наша контрольная точка.

Последний шаг — просто обучить сеть и позволить нашей контрольной точке позаботиться обо всем остальном:

```
43 # обучаем сеть
44 print("[INFO] обучайся сеть...")
45 H = model.fit(trainX, trainY, validation_data=(testX, testY),
46     batch_size=64, epochs=40, callbacks=callbacks, verbose=2)
```

Чтобы выполнить наш скрипт, просто откройте терминал и выполните следующую команду:

```
$ python cifar10_checkpoint_improvements.py --веса в весов/улучшений [ИНФО]
загрузка данных CIFAR-10...
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 50000 выборках, проверка на 10000
выборках Эпох а 1/40 171 с - потеря 1,6700 - акк: 0,4375
- val_loss: 1,2697 - val_acc: 0,5425 Эпох а 2/40 Эпох а 00001: val_loss улучшен с
1,26973 до 0,98481, сохранение модели для тестирования/
```

```
веса-001-0.9848.hdf5
...
Эпох а 40/40
Эпох а 00039 : val_loss не улучшилось 315 с
- потеря 0,2594 - акк: 0,9075 - val_loss: 0,5707 - val_acc: 0,8190
```

```

315s - loss: 0.3161 - acc: 0.8867 - val_loss: 0.5704 - val_acc: 0.8175
Epoch 31/40
Epoch 00030: val_loss did not improve
318s - loss: 0.3060 - acc: 0.8899 - val_loss: 0.5649 - val_acc: 0.8188
Epoch 32/40
Epoch 00031: val_loss did not improve
316s - loss: 0.3062 - acc: 0.8907 - val_loss: 0.5640 - val_acc: 0.8207
Epoch 33/40
Epoch 00032: val_loss did not improve
316s - loss: 0.2948 - acc: 0.8947 - val_loss: 0.5738 - val_acc: 0.8158
Epoch 34/40
Epoch 00033: val_loss improved from 0.56279 to 0.55456, saving model to test/weights-033-0.5546.hdf5
316s - loss: 0.2949 - acc: 0.8940 - val_loss: 0.5546 - val_acc: 0.8218
Epoch 35/40
Epoch 00034: val_loss did not improve
314s - loss: 0.2854 - acc: 0.8969 - val_loss: 0.5704 - val_acc: 0.8187
Epoch 36/40
Epoch 00035: val_loss did not improve
316s - loss: 0.2836 - acc: 0.8987 - val_loss: 0.5650 - val_acc: 0.8209
Epoch 37/40
Epoch 00036: val_loss did not improve
315s - loss: 0.2697 - acc: 0.9046 - val_loss: 0.5669 - val_acc: 0.8204
Epoch 38/40
Epoch 00037: val_loss did not improve
316s - loss: 0.2680 - acc: 0.9044 - val_loss: 0.5711 - val_acc: 0.8219
Epoch 39/40
Epoch 00038: val_loss did not improve
316s - loss: 0.2634 - acc: 0.9041 - val_loss: 0.6066 - val_acc: 0.8096
Epoch 40/40
Epoch 00039: val_loss did not improve
315s - loss: 0.2594 - acc: 0.9075 - val_loss: 0.5707 - val_acc: 0.8190

```

Рисунок 18.1: Проверка отдельных моделей каждый раз, когда продолжительность модели улучшается, что приводит к созданию нескольких файлов веса после завершения обучения

Как видно из вывода моего терминала и рисунка 18.1, каждый раз, когда потери при проверке уменьшаются мысох раннейновую сериализованную модель на диск.

В конце процесса обучения у нас есть 18 отдельных файлов, по одному для каждого постепенного улучшения:

```
$ найти ./ -printf "%f\n" | сортировка .
веса-000-1.2697.hdf5
веса-001-0.9848.hdf5
веса-003-0.8176.hdf5
веса-004-0.7987.hdf5
веса-005-0.7722.hdf5
веса-006-0.6925.hdf5
веса-007-0.6846.hdf5
веса-008-0.6771.hdf5
веса-009-0.6212.hdf5
веса-012-0.6121.hdf5
веса-013-0.6101.hdf5
веса-014-0.5899.hdf5
веса-015-0.5811.hdf5 веса-017-0.5774
hdf5 веса-019-0.5740.hdf5
веса-022-0.5724.hdf5
```

```
веса-024-0.5628.hdf5
веса-033-0.5546.hdf5
```

Как видите, каждое имя файла состоит из трех компонентов. Первая — это статическая строка, веса. Затему нас есть номер эпохи. Последний компонент имени файла — это метрика, которую мы измеряем для улучшения в данном случае это потеря проверки.

Наша лучшая потеря при проверке была получена в эпохе 33 с значением 0,5546. Затем мы могли бы взять эту модель и загрузить ее с диска (глава 13), а затем оценить ее или применить к нашим собственным изображениям (о чем мы поговорим в следующей главе).

Имейте в виду, что ваши результаты будут совпадать с моими, поскольку сети являются состоящими и инициализируются случайными величинами. В зависимости от начальных значений у вас могут быть совершенно разные контрольные точки модели, в конце процесса обучения наши сети должны получить одинаковую точность (\pm несколько процентных пунктов).

18.2 Проверка только лучшей нейронной сети

Возможно, самым большим недостатком постепенных улучшений с контрольными точками является то, что мы получаем кучу дополнительных файлов, которые нас (маловероятно) интересуют, что особенно верно, если наши потери при проверке перемещаются вверх и вниз в течение эпох обучения — каждое из этих дополнительных улучшений будет быть захвачены и сериализованы на диск. В этом случае лучше в сего со временем обновлять только одну модель и просто перезаписывать ее каждый раз, когда наша метрика улучшается во время обучения.

К счастью, выполнить это действие так же просто, как обновить класс `ModelCheckpoint`, чтобы он принимал простую строку (т. е. путь к файлу без каких-либо переменных шаблона). Затем, когда наша метрика улучшается этот файл просто перезаписывается. Чтобы понять процесс, давайте создадим второй файл Python с именем `cifar10_checkpoint_best.py` и рассмотрим различия.

Во-первых, нам нужно импортировать необязательные пакеты Python:

```
1 # импортируйте необязательные пакеты
2 из sklearn.preprocessing import LabelBinarizer 3 из
pyimagesearch.nn.conv import MiniVGGNet 4 из keras.callbacks
import ModelCheckpoint 5 из keras.optimizers import SGD 6 из
keras.datasets import cifar10 7 из argparse
```

Затем проанализируйте наши аргументы командной строки:

```
9 # построить разбор аргумента и разобрать аргументы 10 ap =
argparse.ArgumentParser() 11 ap.add_argument("-w", "--weights",
required=True,
12     help="путь к файлу лучших весов моделей")
13 args = vars(ap.parse_args())
```

Имя самого аргумента командной строки тоже (`--weights`), но описание ключа теперь другое: «путь к файлу лучших весов моделей». Таким образом, этот аргумент командной строки будет простой строкой для ввода одного пути — к этой строке не будет применено никакого обзора.

Отсюда мы можем загрузить наш набор данных CIFAR-10 и подготовить его для обучения

```

15 # загрузить данные обучения и тестирования, затем масштабировать их
16 # [0, 1] 17 print("[INFO] загрузка данных CIFAR-10...")
18 ((trainX,
trainY), (testX, testY)) = cifar10.load_data() 19 trainX = trainX.astype("float") / 255.0 20
testX = testX.astype("float") / 255.0
[REDACTED]
21
22 # преобразовать метки из целых чисел в векторы 23
lb = LabelBinarizer() 24 trainY = lb.fit_transform(trainY) 25 testY =
lb.transform(testY)

```

А также инициализируем наш оптимизатор SGD и архитектуру MiniVGGNet:

```

27 # инициализируем оптимизатор и модель 28
print("[INFO] компилируем модель...")
29 opt =
SGD(lr=0.01, распад=0.01 / 40, импульс=0.9, нестеров=True) 30 model = MiniVGGNet.build
(ширина = 32, высота = 32, глубина = 3, классы = 10) 31
model.compile(loss="categorical_crossentropy", оптимизатор=opt,
метрики=["точность"])
32

```

Теперь мы готовы обучить код ModelCheckpoint:

```

34 # создать обратный вьюв для сохранения на диск только *лучшей* модели 35
# на основе потери проверки
36 контрольная точка = ModelCheckpoint(args["веса"], монитор="val_loss",
37                                         save_best_only=True, verbose=1)
38 обратный вьювов = [контрольная точка]

```

Обратите внимание, что строка шаблона fname исчезла — все, что мы делаем, это передаем значение --weights в ModelCheckpoint. Поскольку нет значений шаблона для заполнения, Keras просто перезапишет существующий сериализованный файл весов всякий раз, когда наша метрика мониторинга улучшается (в данном случае потеря проверки).

Наконец, мы тренируемся сети в блоке кода ниже:

```

40 # обучаем сеть
41 print("[INFO] обучая щасеть...")
42 H = model.fit(trainX, trainY, validation_data=(testX, testY), batch_size=64, epochs
43             =40, callbacks=callbacks, verbose=2)

```

Чтобы выполнить наш скрипт, введите следующую команду:

```

$ python cifar10_checkpoint_best.py --weights test_best/cifar10_best_weights.hdf5 [INFO] загрузка данных
CIFAR-10...
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 50000 выборках, проверка на 10000 выборках
Эпохи 1/40 Эпохи 00000: значение val_loss улучшено с inf до
1,26677, модель сохранена в
test_best/cifar10_best_weights.hdf5

```

305s - потеря 1,6657 - акк: 0,4441 - val_loss: 1,2668 - val_acc: 0,5584 Эпох а 2/40 Эпох а 00001: val_loss улучшен с 1,26677 до 1,21923, сох ранение модели в

```
test_best/cifar10_best_weights.hdf5
309s - убыток: 1,1996 - акк: 0,5828 - val_loss: 1,2192 - val_acc: 0,5798
...
Эпох а 40/40
Эпох а 00039: val_loss не улучшилось 173 с - потеря
0,2615 - акк: 0,9079 - val_loss: 0,5511 - val_acc: 0,8250
```

Здесь вы можете видеть, что мы перезаписываем наш файл cifar10_best_weights.hdf5 обновленным сетью только в том случае, если наши потери при проверке уменьшаются. Это имеет два основных преимущества:

1. В конце процесса обучения есть только один сериализованный файл — модельная эпоха, которая получила наименьшие потери.
2. Мы фиксируем «постепенное улучшение», когда потери колеблются вверх и вниз. Вместо этого мы сохраняем перезаписываемую лучшую модель только в том случае, если наша метрика получает потери меньше, чем в предыдущие эпохи.

Чтобы убедиться в этом, взгляните на мой каталог weights/best, где вы видите только один в нем один файл:

```
$ ls -l weights/best/
total 17024
-rw-rw-r-- 1 adrian adrian
17431968 28 апреля 09:47 cifar10_best_weights.hdf5
```

Затем вы можете взять этот сериализованный MiniVGGNet и дополнительно оценить его на данных тестирования или применить к своим собственным изображениям (описано в главе 15).

18.3 Резюме

В этой главе мы рассмотрели, как отслеживать заданную метрику (например, потери при проверке, точность проверки и т. д.) вовремя обучения, а затем сохранять в высокопроизводительные сети на диск. Есть два способа сделать это внутри Keras: 1. Итерационные улучшения Checkpoint.

2. Проверяйте только лучшую модель, найденную в процессе.
Лично я предпочитаю последнее первому, поскольку оно приводит к меньшему количеству файлов и одному в них одному файлу, который представляет собой лучшую эпоху, найденную в процессе обучения.

19. Визуализация сетевых архитектур

Одной из концепций, которую мы еще не обсуждали, является визуализация архитектуры, процесс построения графа узлов и связанных соединений в сети и сохранение графа на диск в виде изображения (например, .PNG, JPG и т. д.). Узлы на графиках представляют слои, а соединения между узлами представляют поток данных через сеть.

Эти графики обычно включают следующие компоненты для каждого слоя:

1. Размер в \times одного объема.

2. Размер в \times одного тома.

3. И пожеланию имени слоя

Обычно мы используем визуализацию сетевой архитектуры при (1) отладке наших собственных пользовательских сетевых архитектур и (2) публикации, когда визуализацию архитектуры легче понять, чем включать фактический исходный код или пытаться создать таблицу для передачи той же информации. В оставшейся части этой главы вы узнаете, как создавать графы визуализации сетевой архитектуры с помощью Keras, а затем сериализовать график на диск как реальное изображение.

19.1 Важность визуализации архитектуры

Визуализация архитектуры модели является важным инструментом отладки, особенно если вы:

1. Внедряете архитектуру в публикацию, но не знакомы с ней.

2. Внедрение собственной пользовательской сетевой архитектуры

Короче говоря, визуализация сети подтверждает наши предположения о том, что наш код правильно строит модель, которую мы собираемся построить. Изучив изображение в \times одного графика, вы можете увидеть, есть ли ошибка в вашей логике. К наиболее распространенным недостаткам относятся:

- Неправильный порядок слоев в сети.

2. Предположение о (неправильном) размере в \times одного тома после слоя CONV или POOL.

При реализации сетевой архитектуры я предлагаю вам визуализировать сетевую архитектуру после каждого блока слоев CONV и POOL, что позволит вам проверить свои предположения (и, чтобы более важно, выявить «ошибки» в сети на ранней стадии).

Ошибки в сверточных нейронных сетях не похожи на другие логические ошибки в приложениях, возникающие в результате крайних случаев. Вместо этого CNN может очень хорошо обучаться и получать разумные результаты даже при

неправильный порядок слоев, но если вы не понимаете, что произошла эта ошибка, вы можете сообщить об ошибках результатах, думая что сделали одно, а на самом деле сделали другое.

В оставшейся части этой главы помогут вам визуализировать вашу собственную сетевую архитектуру, чтобы избежать подобных проблем в ситуациях.

19.1.1 Установка graphviz и pydot

Чтобы построить график нашей сети и сохранить его на диск с помощью Keras, нам нужно установить предварительное условие graphviz: В Ubuntu это так же просто, как:

```
$ sudo apt-get установить графвиз
```

В macOS мы можем установить graphviz через Homebrew:

```
$ варить установить графвиз
```

После установки библиотеки graphviz нам нужно установить два пакета Python:

```
$ pip install graphviz==0.5.2 $ pip
install pydot-ng==1.0.0
```

Приведенные выше инструкции были включены в главу 6, когда вы настраивали машину для разработки для глубокого обучения, навязав ключи здесь, а также для полноты картинки. Если у вас возникли трудности с установкой этих библиотек, обратитесь к соответствующему дополнительному материалу в конце главы 6.

19.1.2 Визуализация сетей Keras

Визуализация сетей архитектур с помощью Keras невероятно проста. Чтобы увидеть, насколько это просто, откройте новый файл, назовите его `visualize_architecture.py` и вставьте следующий код:

```
1 # импортируем необходимые пакеты
2 из pyimagesearch.nn.conv import LeNet3 из
keras.utils import plot_model
4
5 # инициализируем LeNet, а затем пишем архитектуру сети
6 # визуализация графика на диск
7 model = LeNet.build(28, 28, 1, 10) 8
plot_model(model, to_file="lenet.png", show_shapes=True)
```

Строка 2 импортирует нашу реализацию LeNet (глава 14) — это сетевая архитектура, которую мы будем визуализировать. Страна 3 импортирует функцию `plot_model` из Keras. Как следует из названия этой функции, `plot_model` отвечает за построение графика на основе слоев внутри в одной модели и последующую запись графика на диск с изображением.

В строке 7 мы создаем экземпляр архитектуры LeNet, как если бы мы собирались применить ее к MNIST для классификации цифр. Параметры включают ширину в одногом объеме (28 пикселей), высоту (28 пикселей), глубину (1 канал) и общее количество меток классов (10).

Наконец, строка 8 отображает нашу модель и сохраняет ее на диск под именем `lenet.png`.

Чтобы выполнить наш скрипт, просто откройте терминал и введите следующую команду:

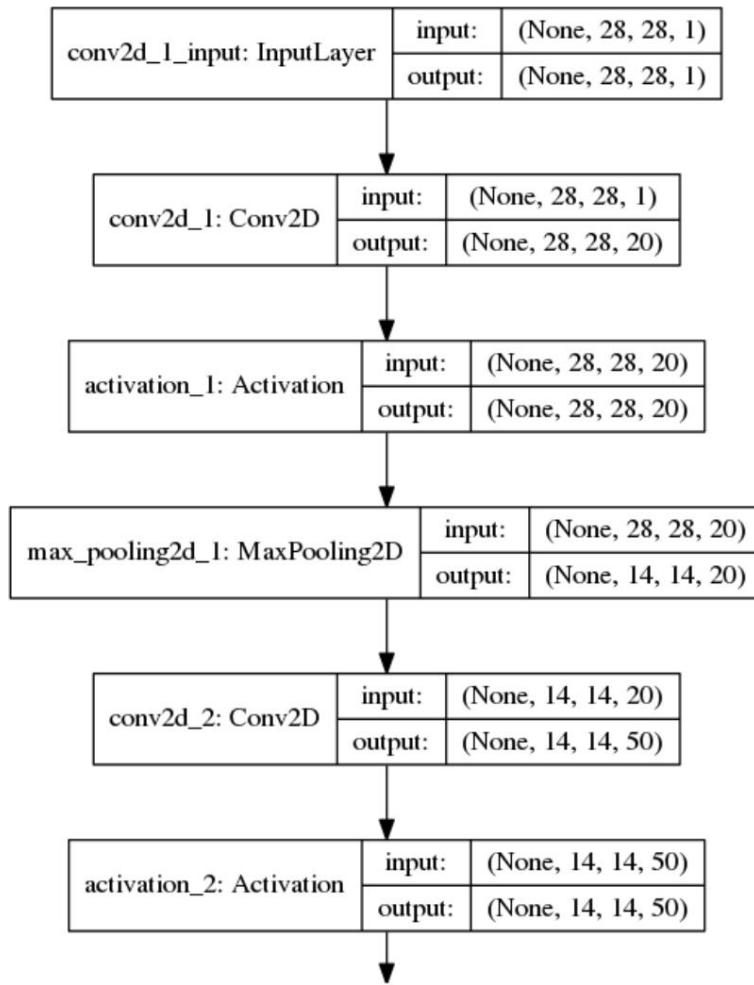


Рисунок 19.1: Часть I графического изображения сети в архитектуре LeNet, созданного Keras. Каждый узел на графике представляет определенную функцию слоя(т. е. свертка, объединение, активация, выравнивание, полносвязность и т. д.). Стрелки представляют поток данных через сеть. Каждый узел также включает размер входа и выхода тома после данной операции.

\$ Python visualize_architecture.py

Как только команда успешно существует, проверьте текущий рабочий каталог:

\$ ls
lenet.png visualize_architecture.py

Как вы видите, есть файл с именем lenet.png — этот файл является нашим фактическим графиком визуализации сети. Откройте его и просмотрите (рис. 19.1 и 19.2).

Здесь мы можем увидеть визуализацию потока данных через нашу сеть. Каждый уровень представлен как узел в архитектуре, который затем подключается к другим уровням в конечном итоге завершается после применения классификатора softmax. Обратите внимание, что каждый слой в сети включает вх однай и вък одной атрибуты — эти значения представляют собой размер пространственных измерений соответствующего объема, когда он входит в слой и после выходит из него.

Пройдя через архитектуру LeNet, мы видим, что первый слой — это наш `InputLayer`, который принимает вх однок изображение $28 \times 28 \times 1$. Пространственные размеры для вода и ввода слоя одинаковы, так как это просто «заполнитель» для вх однок данных.

Вам может быть интересно, что представляет `None` в фигуре данных (`None, 28, 28, 1`). На самом деле `None` — это размер партии. При визуализации сетевой архитектуры Keras не знает предполагаемого размера пакета, поэтому оставляет значение `None`. При обучении это значение изменится на `32, 64, 128` и т. д. или на любой другой размер партии, который мы сочтем подходящим.

Далее наши данные передаются на первый слой `CONV`, где мы изучаем 20 ядер на вх оде $28 \times 28 \times 1$. Вых од этого первого слоя `CONV` составляет $28 \times 28 \times 20$. Мы сошли наши первоначальные пространственные размеры из-за заполнения нулями, но, изучив 20 фильтров, мы изменили размер объема.

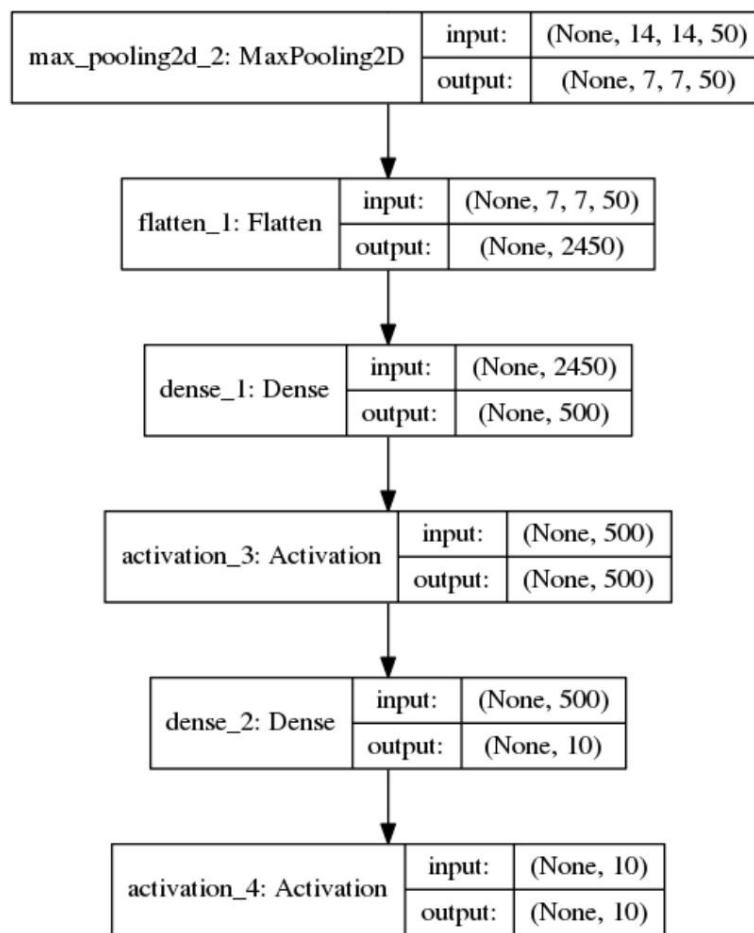


Рисунок 19.2: Часть II визуализации архитектуры LeNet, включая полносвязные слои и классификатор softmax. В этом случае мы предполагаем, что наш экземпляр LeNet будет использоваться с набором данных MNIST, поэтому у нас есть десять вх однок узлов в нашем последнем слое softmax.

Слой активации следует за слоем `CONV`, который по определению не может изменить размер вх одного тома. Однако операция `POOL` может уменьшить размер тома — здесь наш вх одной объема уменьшается с $28 \times 28 \times 20$ до $14 \times 14 \times 20$.

Второй `CONV` принимает объем $14 \times 14 \times 20$ в качестве вх однок данных, но затем изучает 50 фильтров, изменяя размер вх одного объема на $14 \times 14 \times 50$ (опять же, используется заполнение нулями, чтобы гарантировать, что сама свертка не уменьшает ширину и высоту). Активация применяется перед другой

Операция POOL, которая сначала вдвое уменьшает ширину и высоту с $14 \times 14 \times 50$ до $7 \times 7 \times 50$.

На данный момент мы готовы применить наши слои FC. Для этого мы находим в $7 \times 7 \times 50$ список из 2450 значений (поскольку $7 \times 7 \times 50 = 2450$). Теперь, когда мы сладили в вх одни данные сверточной части нашей сети, мы можем применить слой FC, который принимает 2450 в вх значений и изучает 500 узлов. Затем следует активация за которой следует еще один слой FC, на этот раз уменьшая 500 до 10 (общее количество узлов для набора данных MNIST).

Наконец, к каждому из 10 в вх одних узлов применяется классификатор softmax, что дает нам наши окончательные вероятности класса.

19.2 Резюме

Точнее так же, как мы можем выразить архитектуру LeNet в коде, мы также можем визуализировать саму модель в виде изображения. Когда вы начнете свой путь глубокого обучения, я настоятельно рекомендую вам использовать этот код для визуализации любых сетей, с которыми вы работаете, особенно если вы с ними не знакомы.

Убедившись, что вы понимаете поток данных через сеть и то, как размеры тензоров изменяются в зависимости от уровней CONV, POOL и FC, вы получите гораздо более глубокое понимание архитектуры, а не полагайтесь только на код.

При реализации собственных сетевых архитектур я подтверждаю, что они ожидать на правильном пути, визуализируя архитектуру через каждые 2-3 блока слоев, когда я фактически кодирую сеть — это действие помогает мне на них обнаружить ошибки или недостатки в моей логике на раннем этапе.

20. Готовые CNN для классификации

Досих пор мы научились обучать наши собственные сверточные нейронные сети с нуля. Большинство этих CNN были более поверхностными (и на меньших наборах данных), поэтому их можно легко обучить на наших процессорах, не прибегая к более дорогим графическим процессорам, что позволяет нам освоить основы нейронных сетей и глубокого обучения без приходится опустошать наши карманы.

Однако, поскольку мы работали с более мелкими сетями и меньшими наборами данных, мы не смогли воспользоваться всеми преимуществами классификации, которые дает нам глубокое обучение. К счастью, библиотека Keras поставляется с пятью CNN, предварительно обученными на наборе данных ImageNet:

- VGG16
- VGG19
- ResNet50 •

Inception V3 •

Xception Как мы

обсуждали в главе 5, цель конкурса ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [42] состоит в том, чтобы обучить модель, которая может правильно классифицировать вх одное изображение по 1000 отдельных категорий объектов. Эти 1000 категорий изображений представляют собой классы объектов, с которыми мы сталкиваемся в повседневной жизни, например, в виде собак, кошек, различные предметы домашнего обида, типы транспортных средств и многое другое.

Это означает, что если мы используем CNN, предварительно обученные на наборе данных ImageNet, мы можем распознавать все эти 1000 категорий объектов «из коробки» — никакого обучения не требуется! Полный список категорий объектов, которые можно распознать с помощью предварительно обученных моделей ImageNet, можно найти здесь <http://pyimg.co/x1ler>.

В этой главе мы рассмотрим предварительно обученные современные модели ImageNet в библиотеке Keras. Затем я продемонстрирую, как мы можем написать сценарий Python, чтобы использовать эти сети для классификации наших собственных пользовательских изображений без необходимости обучения этих моделей с нуля.

20.1 Современные CNN в Керасе

В этот момент вы вероятно задаетесь вопросом:

«У меня нет дорогостоящего графического процессора. Как я могу использовать эти огромные сети глубокого обучения, которые были предварительно обучены на наборах данных, намного больших, чем те, с которыми мы работали в этой книге?»

Чтобы ответить на этот вопрос, обратитесь к главе 8, посвященной параметризованному обучению. Напомним, что суть параметризованного обучения в том:

1. Определить модель машинного обучения, которая может изучать шаблоны из наших вх однък данных вовремя обучения (требуя, чтобы мы тратили больше времени на процесс обучения), но чтобы процесс тестирования был намного быстрее.
2. Получить модель, которую можно определить с помощью небольшого числа параметров, которые легко представляют сеть, независимо от размера обучения

Следовательно, фактический размер нашей модели зависит от ее параметров, а не от количества обучающих данных. Мы могли бы обучить очень глубокую CNN (такую как VGG или ResNet) на наборе данных из 1 миллиона изображений или наборе данных из 100 изображений, но результатирующий размер вх однък модели будет таким же, поскольку размер модели определяется встроенной нами архитектурой.

Вторым, нейронные сети загружают большую часть работы. Мы тратим большую часть нашего времени на обучение наших CNN, будь то из-за глубины архитектуры, количества обучающих данных или количества экспериментов, которые мы должны провести для настройки наших гиперпараметров.

Оптимизированное оборудование, такое как графические процессоры, позволяет нам ускорить процесс обучения, поскольку нам нужно выполнить как прямой проход, так и обратный проход в алгоритме обратного распространения — как мы уже знаем, именно в этом процессе наша сеть фактически учится. Однако после того, как сеть обучена, нам нужно только выполнить прямой проход, чтобы классифицировать данное вх однъе изображение. Прямой проход однозначно быстрее, что позволяет нам классифицировать вх однъе изображения с помощью глубоких нейронных сетей на ЦП.

В большинстве случаев сетевые архитектуры, представленные в этой главе, не смогут обеспечить реальную производительность в реальном времени на ЦП (для этого нам понадобится ГП) — но это нормально; вы по-прежнему можете использовать эти предварительно обученные сети в своих собственных приложениях. Если вы заинтересованы в том, чтобы научиться обучать современные сверточные нейронные сети с нуля на сложном наборе данных ImageNet, обязательно обратитесь к пакету ImageNet этой книги, где я демонстрирую именно это.

20.1.1 VGG16 и VGG19

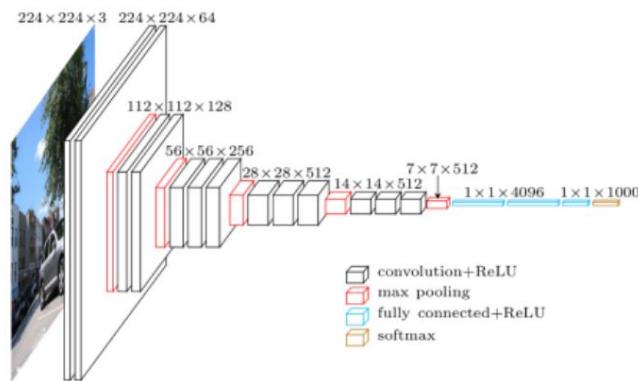


Рисунок 20.1: Визуализация архитектуры VGG. В сеть вводят изображение размером $224 \times 224 \times 3$. Затем применяются фильтры свертки только 3×3 с большим количеством сверток, расположенных друг на друга перед максимальным объемом единением операций глубже в архитектуре. Изображение предоставлено: <http://pyimg.co/xgiel>

Сетевая архитектура VGG (рис. 20.1) была представлена Симоном и Зиссерманом в их статье 2014 года «Очень глубокие сверточные сети для крупномасштабного распознавания изображений» [95].

Как мы обсуждали в главе 15, семейство сетей VGG характеризуется использованием только сверточных слоев 3×3 , расположенных друг на друга с возрастанием глубины. Уменьшение размера тома осуществляется с помощью максимального объединения. Затем из двух полносвязанных слоев по 4096 узлов следует классификатор softmax.

В 2014 году 16- и 19-уровневые сети считались очень глубокими, хотя сейчас у нас есть архитектура ResNet, которую можно успешно обучать на глубинах 50-200 для ImageNet и более 1000 для CIFAR-10. К сожалению, у VGG есть два основных недостатка:

1. Обучение проходит мучительно медленно (к счастью, в этой главе мы тестируем только ковровые однотипные изображения).
2. Сама веса сети значительно больше (с точки зрения дискового пространства/пропускной способности). Из-за глубины количества полностью подключенных узлов сериализованное файловое дерево для VGG16 составляет 533 МБ, а для VGG19 — 574 МБ.

К счастью, эти веса нужно загрузить только один раз — оттуда мы можем кэшировать их на диск.

20.1.2 Резет

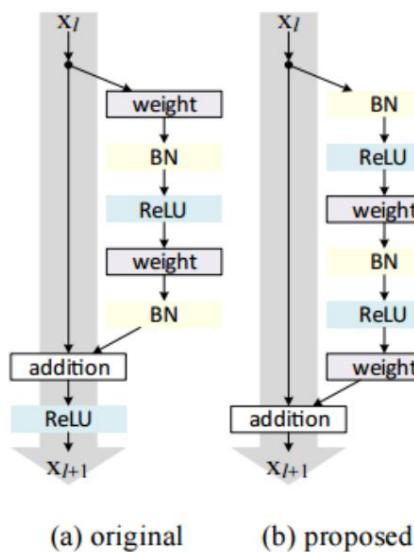


Рисунок 20.2: Слева: Исходный остаточный модуль. Справа: обновленный остаточный модуль с использованием предварительной активации. Рисунки из He et al., 2016 [130].

В первые представления He et al. в их статье 2015 года «Глубокое остаточное обучение для распознавания изображений» [96] архитектура ResNet стала основой для работы в литературе по глубокому обучению, продемонстрировав, что с помощью глубоких сетей можно обучать с использованием стандартного SGD (и разумной функции инициализации) через использование остаточных модулей.

Дополнительную точность можно получить, обновив остаточный модуль для использования отображений идентичности (рис. 20.2), как показано в их последней публикации 2016 года «Отображение идентичности в глубоких остаточных сетях» [130].

Тем не менее, имейте в виду, что реализация ResNet50 (как в 50 весовых слоях) в основной библиотеке Keras основана на бывшей статье 2015 года. Несмотря на то, что ResNet намного глубже, чем VGG16 и VGG19, размер модели на самом деле значительно меньше из-за использования глобального среднего пула, а не полносвязных слоев, что уменьшает размер модели до 102 МБ для ResNet50.

Если вы хотите узнать больше об архитектуре ResNet, включая основной модуль и то, как он работает, обратитесь к пакетам Practitioner Bundle и ImageNet Bundle, в которых подробно рассматривается ResNet.

20.1.3 Начальная версия V3

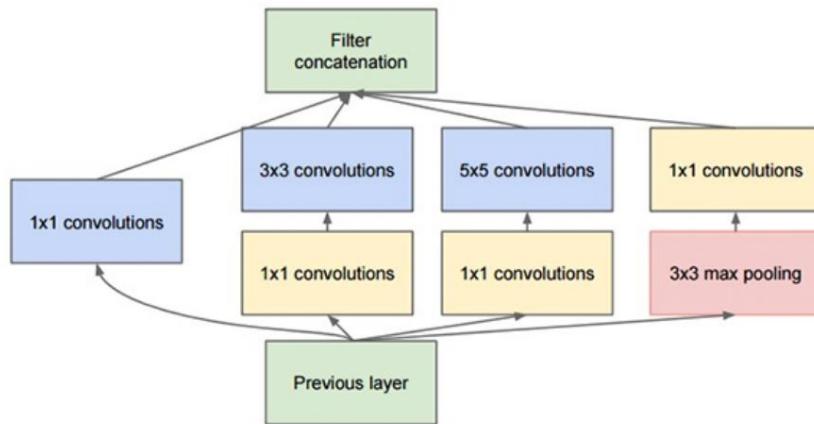


Рисунок 20.3: Исходный модуль Inception, используемый в GoogLeNet. Модуль Inception действует как «многоуровневый экстрактор признаков», в нем используются свертки 1×1 , 3×3 и 5×5 в одном и том же модуле сети. Рисунок из Szegedy et al., 2014 [97].

Модуль «Inception» (и полученная в результате архитектура Inception) был представлен Szegedy et al. их статья 2014 года Going Deeper with Convolutions [97]. Цель начального модуля (рис. 20.3) — действовать как «многоуровневый экстрактор признаков», в нем используются свертки 1×1 , 3×3 и 5×5 в одном и том же модуле сети. Затем складываются по размеру канала перед подачей на следующий уровень в сети.

Первоначальное описание этой архитектуры называлось GoogLeNet, но последующие вложения были просто названы Inception vN, где N означает номер версии, выпущенной Google. Архитектура Inception V3, включенная в ядро Keras, исходит из более поздней публикации Szegedy et al., Rethinking the Inception Architecture for Computer Vision (2015) [131], в которой предлагаются обновления начального модуля для дальнейшего повышения точности классификации ImageNet. Вес Inception V3 меньше, чем у VGG и ResNet, и составляет 96 МБ.

Для получения дополнительной информации о том, как работает модуль Inception (и как обучать GoogLeNet из нуля), пожалуйста, обратитесь к комплекту Practitioner Bundle и ImageNet Bundle.

20.1.4 Xception Xception

было предложение кем-нибудь, как самим Франсуа Шалле, создателем и главным автором библиотеки Keras, в его статье 2016 года Xception: Deep Learning with Depthwise Separable Convolutions [132]. Xception — это расширение архитектуры Inception, которое заменяет стандартные модули Inception на разделение по глубине свертки. Веса Xception являются самыми меньшими из предварительно обученных сетей, включенных в библиотеку Keras, и весят 91 МБ.

20.1.5 Может ли мы стать меньше?

Xception не включена в библиотеку Keras, ях отел упомянуть, что архитектура SqueezeNet [127] часто используется, когда нам нужна небольшая площадь. SqueezeNet очень мал, всего 4,9 МБ.

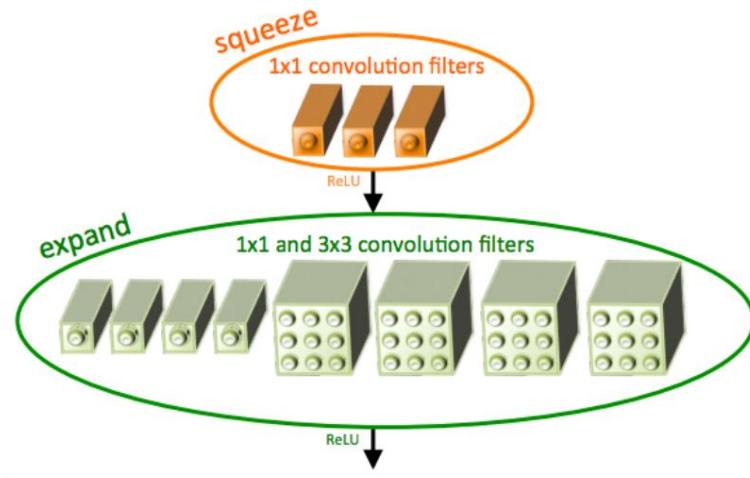


Рисунок 20.4: Модуль «огонь» в SqueezeNet, состоящий из «сжатия» и «расширения». Рисунок из Iandola et al, 2016 [127].

часто используется когда сети необх одимо обучить, а затем развернуть по сети и/или на устройствах с ограниченными ресурсами.

Опять же, SqueezeNet не включен в ядро Keras, но я демонстрирую, как его обучать из подарапать набор данных ImageNet внутри пакета ImageNet.

20.2 Классификация изображений с предварительно обученными CNN ImageNet

Давайте узнаем, как классифицировать изображения с помощью предварительно обученных сверточных нейронных сетей с использованием библиотеки Keras. Нам не нужно обновлять наш основной модуль `pyimagesearch`, который мы разрабатывали до сих пор, поскольку предварительно обученные модели уже являются частью библиотеки Keras.

Просто откройте новый файл, назовите его `imagenet_pretrained.py` и вставьте следующий код:

```
1 # импортируем необх одимые пакеты
из keras.applications импортируем ResNet50 3 из
keras.applications импортируем InceptionV3 4 из
keras.applications импортируем Xception # ТОЛЬКОTensorFlow 5 из
keras.applications импортируем VGG16 6 из keras.applications импортируем
VGG19 7 из keras.applications импортируем imagenet_utils 8 из
keras.applications.inception_v3 import preprocess_input 9 из
keras.preprocessing.image import img_to_array 10 из keras.preprocessing.image
import load_img 11 import numpy as np 12 import argparse 13 import cv2
```

Строки 2-13 импортируют необх одимые пакеты Python. Как видите, большинство пакетов являются частью библиотеки Keras. В частности, строки 2-6 обрабатывают импорт реализаций Keras для ResNet50, Inception V3, Xception, VGG16 и VGG19 соответственно. Обратите внимание, что сеть Xception совместима только с бэкендом TensorFlow (класс выдаст ошибку, если вы попытаетесь создать его экземпляр при использовании бэкенда Theano).

Строка 7 дает нам доступ к подмодулю `imagenet_utils`, удобному набору функций, которые упростят предварительную обработку наших вх однок изображений и расшифровку вх однок классификаций. Остальная часть импорта — это другие вспомогательные функции, за которыми следует NumPy для числовых функций. Operations и cv2 для наших привязок OpenCV.

Далее, давайте проанализируем наши аргументы командной строки:

```
15 # построить аргумент parse и разобрать аргументы
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-i", "--image", required=True,
18                  help="путь к вх одному изображению")
19 ap.add_argument("-model", "--model", type=str, default="vgg16",
20                  help="имя предварительно обученной сети для использования")
21 аргумент = варс(ap.parse_args())
```

Нам потребуется только один аргумент командной строки, `--image`, который является путем к нашему вх оду. изображение, которое мы хотим классифицировать. Мы также применим обязательный аргумент командной строки `-model`, строка, указывающая какую предварительно обученную CNN мы хотели бы использовать — это значение по умолчанию равно `vgg16` для Архитектура VGG16.

Учитывая что мы принимаем имя предварительно обученной сети через аргумент командной строки, нам нужно чтобы определить словарь Python, который сопоставляет имена моделей (строки) с их фактическими классами Keras:

```
23 # определить словарь, который сопоставляет имена моделей с их классами
24 # внутри Кераса
25 МОДЕЛЕЙ = {
26     "vgg16": VGG16,
27     "vgg19": VGG19,
28     "начало": InceptionV3,
29     "xception": Xception, # ТОЛЬКО TensorFlow
30     "реснет": ResNet50
31 }
32
33 # убедитесь, что правильное имя модели было указано через аргумент командной строки
34 , если args["model"] отсутствует в MODELS.keys():
35     поднять AssertionError("Аргумент командной строки --model должен быть ключом в
36     словаре 'MODELS'")
```

Строки 25-31 определяют наш словарь `MODELS`, который сопоставляет строку имени модели с соответствующим классом. Если имя `-model` не найдено внутри `MODELS`, мы выбросим `AssertionError`. (строки 34-36).

Как мы уже знаем, CNN принимает изображение в качестве вх однок `данных`, а затем возвращает набор векторов. соответствующие меткам класса в качестве вх однок `данных`. Типичные размеры вх однок изображений для CNN, обученной на ImageNet 224×224, 227×227, 256×256 и 299×299; однаково можете видеть другие измерения как х орошо.

VGG16, VGG19 и ResNet принимают вх однок изображения 224×224, в то время как Inception V3 и Для Xception требуется вх однок размером 229×229 пикселей, как показано в следующем блоке кода:

```
38 # инициализируем форму вх одного изображения (224x224 пикселя) вместе с
39 # функция предварительной обработки (может потребоваться изменить
40 # в зависимости от того, какую модель мы используем для классификации нашего изображения)
41 inputShape = (224, 224)
```

```

42 preprocess = imagenet_utils.preprocess_input
43
44 # если мы используем сети InceptionV3 или Xception, то нам 45 # нужно
установить вх одную форму на (299x299) [а не (224x224)] 46 # и использовать
другую функцию обработки изображения 47 if args["model"] in ("начало",
"исключение"): inputShape = (299, 299) preprocess = preprocess_input
48
49

```

Здесь мы инициализируем нашу inputShape размером 224×224 пикселя. Мы также инициализируем нашу функцию предварительной обработки, чтобы она была стандартной функцией preprocess_input от Keras (которая включает в себя введение среднего, метод нормализации, который мы рассматриваем в пакете Practitioner Bundle). Однако, если мы используем Inception или Xception, нам нужно установить inputShape на 299×299 пикселей, а затем обновить предварительную обработку, чтобы использовать отдельную функцию предварительной обработки, которая включает в себя другой тип масштабирования <http://pyimg.co/3ico2>.

Следующим шагом является загрузка весов нашей предварительно обученной сети в архитектуре с диска и создание экземпляра нашей модели:

```

51 # загружаем весы коэффициенты с диска (ПРИЧАНИЕ: если
это 52 # первый раз, когда запускаете этот скрипт для данной сети, сначала
нужно будет загрузить 53 # весы коэффициенты - в зависимости от того, в
какой 54 # сети вы находитесь при использовании, веса могут быть 90-575 МБ,
так что будьте 55 # терпеливы: веса будут кэшироваться и последующие
запуски этого 56 # скрипта будут *много* быстрее) 57 print("[INFO] loading
{...} .формат(аргумент["модель"]))
58 Сеть = МОДЕЛИ[аргумент["модель"]]
59 модель = Сеть(веса="изображение")

```

В строке 58 используется слово MODEL вместе с аргументом командной строки --model для получения правильного класса сети. Затем CNN создается в строке 59 с использованием предварительно обученных весов ImageNet.

Отметьте же, имейте в виду, что веса для VGG16 и VGG19 составляют 500 МБ. Веса ResNet составляют 100 МБ, а веса Inception и Xception — от 90 до 100 МБ. Если вы впервые запускаете этот сценарий для данной сетевой архитектуры, эти веса будут (автоматически) загружены в кешевый диск. В зависимости от скорости вашего интернета это может занять некоторое время. Однако после загрузки весов их не нужно будет загружать снова, что позволяет выполнять последующие запуски imagenet_pretrained.py намного быстрее.

Теперь наша сеть загружена и готова к классификации изображения — нам просто нужно подготовить изображение к классификации, предварительно обработав его.

```

61 # загрузите вх одное изображение с помощью вспомогательной утилиты Keras,
убедившись, что 62 # размер изображения изменен на 'inputShape', требуемое
вх одные размеры 63 # для предварительно обученной сети ImageNet 64 print("[INFO]
загрузка и предварительная обработка изображения...") 65 image =
load_img(args["image"], target_size=inputShape) 66 image = img_to_array(image)

```

67

```

68 # наше вх одное изображение теперь представлено в виде массива NumPy
# формы 69 # (inputShape[0], inputShape[1], 3) однако нам нужно расширить размерность
70 #, создав фигуру (1, inputShape[0], inputShape[1], 3) 71 # чтобы мы могли передать его
на сеть

```

```

72 изображение = pr.expand_dims(изображение, ось = 0)
73
74 # предварительно обработать изображение с помощью соответствующей функции на основе
75 # загруженной модели (например, вычитание среднего, масштабирование и т. д.) 76 image =
    preprocess(image)

```

Строка 65 загружает наше вх одное изображение с диска, используя предоставленный inputShape для изменения ширины в высоту изображения. Предполагая что мы используем порядок «каналы в последнюю очередь», наше вх одное изображение теперь представлено в виде массива NumPy с формой (inputShape[0], inputShape[1], 3).

Однако мы обучаем/классифицируем изображения партиями с помощью CNN, поэтому нам нужно добавить дополнительное измерение в массив с помощью функции pr.expand_dims в строке 72. После возврата pr.expand_dims наше изображение теперь будет иметь форму (1, inputShape[0], inputShape[1], 3), оговаривая же, при предположении, что каналы упорядочиваются последними. Если вы забудете добавить это дополнительное измерение, это приведет к ошибке при возврате метода .predict модели.

Наконец, в строке 76 возврашается соответствующая функция предварительной обработки для выполнения вычитания среднего и/или масштабирования.

Теперь мы готовы передать наше изображение через сеть и получить вх однье классификации:

```

78 # классифицируем
изображение 79 print("[INFO] классифицируем изображение с
'{}'...'.format(args["model"])) 80 preds = model.predict(image)
81 P = imagenet_utils.decode_predictions(preds)
82
83 # перебираем прогнозы от обрываем прогнозы ранга 5 + 84 # в вероятности
на наш терминал 85 for (i, (imagenetID, label, prob)) в enumerate(P[0]):
    print("{}: {:.2f}%".format(i + 1, label, prob * 100))

```

Возврат .predict в строке 80 возвращает прогнозы CNN. Получив эти прогнозы мы передаем их в служебную функцию ImageNet .decode_predictions, чтобы получить список идентификаторов меток классов ImageNet, «удобочитаемых» меток и вероятности, связанный с каждой меткой класса. Затем первые 5 прогнозов (т. е. метки с наибольшей вероятностью) выводятся на наш терминал в строках 85 и 86.

Наши последний блок кода будет обрабатывать загрузку нашего изображения с диска через OpenCV, рисование предсказания № 1 на изображении и, наконец, отображение его на нашем экране:

```

88 # загружаем изображение через OpenCV, рисуем вверхний прогноз на
изображении, 89 # и возвращаем изображение на наш экран 90 orig =
cv2.imread(args["image"]) 91 (imagenetID, label, prob) = P[0][0] 92 cv2.putText(orig,
"Метка: {}".format(метка), (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
94 cv2.imshow("Классификация", orig.) 95 cv2.waitKey(0)
93

```

Чтобы увидеть наши предварительно обученные сети ImageNet в действии, давайте перейдем к следующему разделу.

20.2.1 Результаты классификации

Чтобы классифицировать изображение с помощью предварительно обученной сети Keras, просто используйте наш скрипт imagenet_retrained.py, а затем выполните (1) путь к вх одному изображению, которое вы хотите классифицировать, и (2) имя сетевой архитектуры, которую вы хотите классифицировать, использовать.

Я включил примеры команд для каждой из доступных предварительно обученных сетей, доступных в Keras ниже:

```
$ Python imagenet_pretrained.py \
    --image example_images/example_01.jpg --model vgg16
$ Python imagenet_pretrained.py \
    --image example_images/example_02.jpg --model vgg19
$ python imagenet_pretrained.py \
    --image example_images/example_03.jpg --начальная модель $
python imagenet_pretrained.py \
    --image example_images/example_04.jpg \
    --model xception $ python imagenet_pretrained.py \
    --image example_images/example_05.jpg -model resnet
```

На рис. 20.5 ниже показан монтаж результатов, полученных для различных входных изображений. В каждом случае метка, предсказанная данной сетевой архитектурой, точно отражает содержимое изображения.



Рисунок 20.5: Результаты применения различных предварительно обученных сетей ImageNet к входным изображениям. В каждом из примеров предварительно обученная сеть возвращает правильные классификации.

20.3 Резюме

В этой главе мы рассмотрели пять сверточных нейронных сетей, предварительно обученных на наборе данных ImageNet в библиотеке Keras: 1. VGG16.

2. VGG19

3. ResNet50 4.

Inception V3 5.

Xception Затем мы

узнали, как использовать каждую из этих архитектур для классификации собственных входных изображений.

Учитывая, что набор данных ImageNet состоит из 1000 популярных категорий объектов, с которыми вы, вероятно, столкнетесь в повседневной жизни, эти модели являются отличными классификаторами «общего назначения». В зависимости от вашей собственной мотивации и конечных целей изучения глубокого обучения одних этих сетей может быть достаточно для создания желаемого приложения.

Тем не менее, читателям, которые заинтересованы в изучении более продвинутых методов обучения более глубоким сетевым работам на больших наборах данных, я настоятельно рекомендую вам ознакомиться с пакетом для практиков . Для читателей, которые хотят получить полный опыт и узнать, как обучать эти современные сети на сложном наборе данных ImageNet, обратитесь к ImageNet Bundle.

21. Практический пример: взлом капчи с помощью CNN

До сих пор в этой книге мы работали с наборами данных, которые были предварительно скомпилированы и помечены для нас, но что, если мы хотим создать собственный набор данных, а затем обучить на нем CNN? В этой главе я представлю полное тематическое исследование глубокого обучения, которое даст вам пример:

1. Загрузка набора изображений.
2. Маркировка и аннотирование изображений для обучения.
3. Обучение CNN на вашем пользовательском наборе данных.
4. Оценка и тестирование обученной CNN.

Набор данных изображений, которые мы будем загружать, представляет собой набор изображений с капчей, используемых для предотвращения автоматической регистрации ботов или входа на данный веб-сайт (или, что еще хуже, попытки взлома чьей-либо учетной записи).

После того, как мы загрузили набор изображений капчи, нам нужно будет вручную пометить каждую цифру в капче. Как мы знаем, получение и маркировка набора данных может быть половиной (если не больше) дела. В зависимости от того, сколько данных вам нужно, насколько легко их получить и нужно ли вам маркировать данные (т. время и/или финансы (если вы платите кому-то еще за маркировку данных)).

Поэтому по возможности мы стараемся использовать традиционные методы компьютерного зрения для ускорения процесса маркировки. В контексте этой главы, если бы мы использовали программное обеспечение для обработки изображений, такое как Photoshop или GIMP, для ручного извлечения цифр из изображения CAPTCHA для создания нашей обучающей выборки, нам потребовались бы дни непрерывной работы, чтобы выполнить эту задачу.

Однако, применяя некоторые базовые методы компьютерного зрения, мы можем загрузить и пометить наш обучающий набор менее чем за час. Это одна из многих причин, по которой я призываю практиков глубокого обучения также инвестировать в свое образование в области компьютерного зрения. Такие книги, как «[Practical Python](#)» и «[OpenCV](#)», предназначены для того, чтобы помочь вам быстро освоить основы компьютерного зрения и OpenCV — если вы серьезно относитесь к освоению глубокого обучения, применяемого к компьютерному зрению, вам следует изучить основы более широкого компьютерного зрения и изображения. Поле обработки тоже.

Я также хотел бы отметить, что наборы данных в реальном мире не похожи на эталонные наборы данных, такие как MNIST, CIFAR-10 и ImageNet, где изображения аккуратно помечены и организованы, и наша цель состоит только в том, чтобы обучить модель на данных и оценить это. Эти эталонные наборы данных могут быть

сложно, но в реальном мире борьба часто заключается в получении самих (помеченных) данных — и во многих случаях помеченные данные стоят намного больше, чем модель глубокого обучения, полученная в результате обучения сети на вашем наборе данных.

Например, если вы управляете компанией, отвечающей за создание пользовательской системы автоматического распознавания номерных знаков (ANPR) для правительства США, вы можете потратить годы на создание надежного массивного набора данных, в то же время оценивая различные подходы к глубокому обучению для распознавания номерных знаков. Накопление такого массивного размеченного набора данных даст вам конкурентное преимущество перед другими компаниями, и в этом случае сами данные стоят больше, чем конечный продукт.

Ваша компания, скорее всего, будет приобретена просто из-за эксклюзивных прав, которые вы имеете на массивный, помеченный набор данных. Создание потрясающей модели глубокого обучения для распознавания номерных знаков только увеличит ценность вашей компании, но опять же, помеченные данные дорого получать и воспроизводить, поэтому, если у вас есть ключи к набору данных, который трудно (если не невозможно) воспроизвести, не заблуждайтесь: основным активом вашей компании являются данные, а не глубокое обучение.

В оставшейся части этой главы мы рассмотрим, как мы можем получить набор данных изображений, пометить их, а затем применить глубокое обучение для взлома системы CAPTCHA.

21.1 Взлом капчи с помощью CNN

Эта глава разбита на несколько частей, чтобы упростить ее читабельность. В первом разделе я расскажу о наборе данных капчи, с которым мы работаем, и обсудим концепцию ответственного раскрытия информации — то, что вы всегда должны делать, когда речь идет о компьютерной безопасности.

Оттуда я обсуждаю структуру каталогов нашего проекта. Затем мы создаем скрипт Python для автоматически загружать набор изображений, которые мы будем использовать для обучения и оценки.

После загрузки наших изображений нам понадобится немного компьютерного зрения, чтобы помочь нам в маркировке изображений, что делает процесс намного проще и значительно быстрее, чем просто обрезка и маркировка в программах для обработки фотографий, таких как GIMP или Photoshop. Как только мы пометим наши данные, мы обучим архитектуру LeNet — как мы узнаем, мы можем взломать систему проверки и получить 100% точность менее чем за 15 эпох.

21.1.1 Примечание об ответственном раскрытии информации

Живя в северо-восточной/среднезападной части США, трудно путешествовать по основным автомагистралям без E-ZPass [133]. E-ZPass — это электронная система сбора платы за проезд, используемая на многих мостах, автомагистралях и туннелях. Путешественники просто покупают транспондер E-ZPass, размещают его на лобовом стекле своего автомобиля и наслаждаются возможностью быстро проезжать через дорожные сборы без остановок, поскольку любые дорожные сборы взимаются с кредитной карты, прикрепленной к их учетной записи E-ZPass.

E-ZPass сделал дорожные сборы гораздо более «приятным» процессом (если есть такая вещь). Вместо того, чтобы стоять в бесконечных очередях, где необходимо совершить физическую транзакцию (например, отдать кассиру деньги, получить сдачу, получить распечатанный чек для возмещения и т. д.), вы можете просто промчаться по скоростной полосе, не останавливаясь — это экономит кучу времени в путешествии и доставляет гораздо меньше хлопот (хотя вам все равно придется платить за проезд).

Я провожу большую часть своего времени, путешествуя между Мэрилендом и Коннектикутом, двумя штатами вдоль коридора I-95 Соединенных Штатов. Корridor I-95, особенно в Нью-Джерси, содержит множество пунктов взимания платы, поэтому пропуск E-ZPass был для меня легким решением. Около года назад срок действия кредитной карты, которую я привязал к своей учетной записи E-ZPass, истек, и мне нужно было обновить ее. Я зашел на веб-сайт E-ZPass в Нью-Йорке (штат, в котором я купил свой E-ZPass), чтобы войти в систему и обновить свою кредитную карту, но я остановился как вкопанный (рис. 21.1).

Можете ли вы найти недостаток в этой системе? Их «капча» — это не более чем четыре цифры на простом белом фоне, что представляет собой серьезную угрозу безопасности — у кого-то есть хотя бы базовое компьютерное зрение.

The screenshot shows the 'Account Login Form' for EZ Pass New York. At the top, there's a note: 'Please choose the Login Type and enter the information in the Login box. Other required fields are marked with *.' Below this, the 'Login Type' section has three options: 'Account Number' (selected), 'Tag Number (11 numbers beginning with 0)', and 'User Name'. To the right of these are 'Login' and 'Password' fields, both marked with an asterisk (*) indicating they are required. Below the login fields are links for 'New User/Forgot Password?' and 'Forgot User Name?'. In the center, there's a 'Security Message' field containing '9 4 1 9', which is highlighted with a red border. Below it is a 'Refresh!' button. To the right, there's another 'Enter Security Message' field and a 'Logon' button.

Рисунок 21.1: Форма входа в систему EZ Pass New York. Можете ли вы обнаружить недостаток в их системе входа в систему?

или опыт глубокого обучения может разработать часть программного обеспечения, чтобы сломать эту систему.

Именно здесь вступает в действие концепция ответственного раскрытия . Ответственное раскрытие — это термин компьютерной безопасности, описывающий, как раскрыть уязвимость. Вместо того, чтобы публиковать его в Интернете для всеобщего обозрения сразу после обнаружения угрозы, вы пытаетесь сначала связаться с заинтересованными сторонами, чтобы убедиться, что они знают о проблеме. Затем заинтересованные стороны могут попытаться исправить программное обеспечение и устраниТЬ уязвимость.

Простое игнорирование уязвимости и скрытие проблемы является ложной безопасностью, что должно избегать. В идеальном мире уязвимость устраняется до того, как она станет общедоступной.

Однако, когда заинтересованные стороны не признают наличие проблемы или не устраниют ее в разумные сроки, возникает этическая головоломка: скрываете ли вы проблему и делаете вид, что ее не существует? Или вы раскрываете ее, привлекая больше внимания к проблеме, чтобы быстрее ее решить? Ответственное раскрытие информации гласит, что вы сначала доводите проблему до заинтересованных сторон (ответственных) — если она не решена, вам необходимо раскрыть проблему (раскрытие).

Чтобы продемонстрировать, как система E-ZPass NY подвергалась риску, я обучил модель глубокого обучения распознавать цифры в капче. Затем я написал второй скрипт Python, чтобы (1) автоматически заполнить мои учетные данные для входа и (2) взломать капчу, разрешив моему скрипту доступ к моей учетной записи.

В этом случае я только автоматически входил в свою учетную запись. Используя эту «функцию» , я мог автоматически обновлять данные кредитной карты, генерировать отчеты о моих дорожных сборах или даже добавлять новый автомобиль в свою E-ZPass. учетная запись.

Я связался с E-ZPass по электронной почте, телефону и в Твиттере по поводу этой проблемы за год до того , как написал эту главу. Они подтвердили получение моих сообщений; однако ничего не было сделано для решения проблемы, несмотря на многочисленные контакты.

В оставшейся части этой главы я расскажу, как мы можем использовать систему E-ZPass для получения набора данных CAPTCHA, который мы затем пометим и обучим модель глубокого обучения. Я не буду делиться кодом Python для автоматического входа в учетную запись — это выходит за рамки ответственного раскрытия информации, поэтому, пожалуйста , не просите меня предоставить этот код.

Я искренне надеюсь, что к тому времени, когда эта книга будет опубликована, E-ZPass NY обновит свой веб-сайт и устраниТ уязвимость с капчей, таким образом оставив эту главу в качестве отличного примера применения глубокого обучения к набору данных, помеченному вручную, с нулевой угрозой уязвимости. .

Имейте в виду, что со всеми знаниями приходит ответственность. Эти знания ни при каких обстоятельствах не должны использоваться в гнусных или неэтичных целях. Это тематическое исследование существует как метод, демонстрирующий, как получить и пометить пользовательский набор данных с последующим обучением модели глубокого обучения поверх него.

Я должен сказать, что я не несу ответственности за то, как используется этот код — используйте его как

возможность учиться, а не возможность быть гнусным.

21.1.2 Структура каталога разрушителя капчи

Чтобы построить систему разбивания капчи, нам нужно обновить файл `pyimagesearch.utils`. подмодуль и включите новый файл с именем `captcha_helper.py`:

```
|--- pyimagesearch
| |--- __init__.py
| |--- наборы данных
| |--- нн
| |--- предварительная обработка
| |--- утилиты
| | |--- __init__.py
| | |--- captcha_helper.py
```

В этом файле будет храниться служебная функция с именем `preprocess`, которая поможет нам обработать цифры перед подачей . их в нашу глубокую нейронную сеть.

Мы также создадим второй каталог с именем `captcha_breaker` за пределами нашего модуля `pyimagesearch` и включим в него следующие файлы и подкаталоги:

```
|--- captcha_breaker
| |--- набор данных/
| |--- загрузок/
| |--- вывод/
| |--- annotate.py
| |--- download_images.py
| |--- test_model.py
| |--- train_model.py
```

Каталог `captcha_breaker` — это место, где будет храниться весь код нашего проекта для разбивания изображения . капчи. Каталог набора данных — это место, где мы будем хранить наши помеченные цифры, которые мы будем маркировать вручную. Я предпочитаю организовывать свои наборы данных, используя следующий шаблон структуры каталогов:

`корневой_каталог/имя_класса/имя_файла_изображения.jpg`

Поэтому наш каталог набора данных будет иметь структуру:

`набор данных/{1-9}/example.jpg`

Где набор данных — это корневой каталог, {1-9} — возможные имена цифр, а `example.jpg` будет примером данной цифры.

В каталоге загрузок будут храниться необработанные файлы `captcha.jpg`, загруженные с E-ZPass. интернет сайт. Внутри выходного каталога мы будем хранить нашу обученную архитектуру LeNet.

Скрипт `download_images.py`, как следует из названия, будет отвечать за фактически скачивание примеров капч и сохранение их на диск. Как только мы загрузили набор капчи, нам нужно извлечь цифры из каждого изображения и вручную пометить каждую цифру — это будет выполнено с помощью `annotate.py`.

Сценарий `train_model.py` будет обучать LeNet на помеченных цифрах, в то время как `test_model.py` будет применить LeNet к самим изображениям капчи.

21.1.3 Автоматическая загрузка примеров изображений

Первым шагом в создании нашего разбивателя капчи является загрузка самих примеров изображений капчи. Если бы мы щелкнули правой кнопкой мыши изображение капчи рядом с текстом «Изображение безопасности» на рис. 21.1 выше, мы бы получили следующий URL-адрес: <https://www.e-zpassny.com/vector/jcaptcha.do> Если вы скопируете и вставьте этот URL-адрес в свой веб-браузер и несколько раз нажмите «Обновить», вы заметите, что это динамическая программа, которая генерирует новую капчу при каждом обновлении. Поэтому, чтобы получить наш пример изображения капчи, нам нужно запросить это изображение несколько сотен раз и сохранить полученное изображение.

Чтобы автоматически получать новые изображения капчи и сохранять их на диск, мы можем использовать `download_images.py`:

```

1 # импортировать необходимые пакеты 2
импортировать argparse 3 импортировать
запросы 4 импортировать время 5 импортировать
os

6
7 # построить аргумент parse и разобрать аргументы 8 ap =
argparse.ArgumentParser() 9 ap.add_argument("-o", "--output",
required=True, help="путь к выходному каталогу изображений") 11
10     ap.add_argument("-n", "--num-images", type=int,
12         default=500, help="# изображений для загрузки") 13
args = vars(ap.parse_args())

```

Строки 2-5 импортируют необходимые пакеты Python. Библиотека запросов упрощает работу с HTTP-соединениями и широко используется в экосистеме Python. Если в вашей системе еще не установлены запросы, вы можете установить их через:

```
$ pip запросы на установку
```

Затем мы анализируем аргументы командной строки в строках 8-13. Нам потребуется один аргумент командной строки, --output, который представляет собой путь к выходному каталогу, в котором будут храниться наши необработанные изображения капчи (позже мы вручную пометим каждую цифру в изображениях).

Второй необязательный переключатель --num-images управляет количеством изображений капчи, которые мы собираемся загрузить. По умолчанию это значение равно 500 изображениям. Поскольку в каждой капче четыре цифры, это значение 500 даст нам $500 \times 4 = 2000$ цифр, которые мы можем использовать для обучения нашей сети.

Наш следующий блок кода инициализирует URL-адрес изображения капчи, которое мы собираемся загрузить, вместе с общим количеством сгенерированных изображений:

```

15 # инициализируем URL-адрес, содержащий изображения капчи, которые мы будем
16 # загружать вместе с общим количеством загруженных изображений 17 # на данный
момент
18 url = "https://www.e-zpassny.com/vector/jcaptcha.do"
Всего 19 = 0

```

Теперь мы готовы загрузить изображения капчи:

```

21 # цикл по количеству загружаемых изображений 22 для i
в диапазоне (0, args["num_images"]):

```

```

23     попытаться:
24         # попытаться получить новое изображение капчи
25         r = запросы.get(url, время ожидания = 60)
26
27         # сохранить образ на диск
28         p = os.path.sep.join([args["output"], "{}.jpg".format(
29             ул(всего).zfill(5))])
30         f = открыть(p, "wb")
31         f.write(r.content)
32         f.закрыть()
33
34         # обновить счетчик
35         print("[INFO] загружено: {}".format(p))
36         всего += 1
37
38     # обрабатывать, если в процессе загрузки возникают какие-либо исключения
39     кроме:
40         print("[INFO] ошибка загрузки изображения...")
41
42     # вставляем небольшой сон, чтобы быть вежливым с сервером
43     время сна (0,1)

```

В строке 22 мы начинаем перебирать `--num-images`, которые хотим загрузить. Запрос сделано в строке 25 для загрузки изображения. Затем мы сохраняем изображение на диск в строках 28-32. Если там произошла ошибка при загрузке изображения, наш блок `try/except` в строках 39 и 40 перехватывает ее и позволяет нашему сценарию продолжить работу. Наконец, мы вставляем небольшой сон в строку 43, чтобы быть вежливым с сетью. сервер, который мы запрашиваем.

Вы можете выполнить `download_images.py` с помощью следующей команды:

```
$ python download_images.py --выходные загрузки
```

Этот скрипт займет некоторое время, так как мы (1) делаем сетевой запрос на загрузку изображение и (2) вставляли 0,1-секундную паузу после каждой загрузки.

Как только программа завершит выполнение, вы увидите, что ваш каталог загрузки заполнен картинки:

```
$ ls -l загрузки/*.jpg | туалет -л
500
```

Однако это всего лишь необработанные изображения капчи — нам нужно извлечь и пометить каждую из цифр. в капчи, чтобы создать наш обучающий набор. Для этого мы будем использовать немного OpenCV и изображение методы обработки, чтобы сделать нашу жизнь проще.

21.1.4 Аннотирование и создание нашего набора данных

Итак, как вы относитесь к маркировке и аннотированию каждого из наших изображений с капчей? Мы открываем Photoshop или GIMP и используйте инструмент «выделение/выделение», чтобы скопировать заданную цифру, сохранить ее на диск и потом повторять до тошноты? Если бы мы это сделали, нам потребовались бы дни непрерывной работы, чтобы пометить каждую из цифры в необработанных изображениях капчи.

Вместо этого лучшим подходом было бы использование базовых методов обработки изображений внутри OpenCV. библиотека нам в помощь. Чтобы увидеть, как мы можем более эффективно пометить наш набор данных, откройте новый файл, назовите его `annotate.py` и вставив следующий код:

```

1 # импортируем необходимые пакеты
2 из путей импорта imutils
3 импортировать синтаксический анализ
4 импорта imutils
5 импорт cv2
6 импорт ОС
7
8 # построить разбор аргумента и разобрать аргументы
9 ap = argparse.ArgumentParser()
10 ap.add_argument("-i", "--input", required=True,
11                 help="путь к входному каталогу изображений")
12 ap.add_argument("-a", "--annot", required=True,
13                 help="путь к выходному каталогу аннотаций")
14 аргументов = переменные (ap.parse_args())

```

Строки 2–6 импортируют необходимые пакеты Python, а строки 9–14 анализируют нашу командную строку аргументы. Этот скрипт требует два аргумента:

- --input: Входной путь к нашим необработанным изображениям CAPTCHA (т. е. каталог загрузок).
- --annot: Выходной путь к тому месту, где мы будем хранить помеченные цифры (т. е. набор данных каталог).

Наш следующий блок кода получает пути ко всем изображениям в каталоге --input и инициализирует словарь с именем counts , который будет хранить общее количество раз, когда данная цифра (ключ) была помечено (значение):

```

16 # получаем пути к изображениям, затем инициализируем словарь символов
17 # отсчетов
18 imagePaths = список (пути.list_images (аргументы ["ввод"]))
19 отсчетов = {}

```

Фактический процесс аннотации начинается ниже:

```

21 # цикл по путям изображений
22 для (i, imagePath) в перечислении (imagePaths):
23     # показать обновление пользователю
24     print("[INFO] обрабатывает изображение {}/{}/".format(i + 1,
25             len (пути к изображениям)))
26
27     # попробуйте:
28     # загрузите изображение и преобразуйте его в оттенки серого, затем заполните
29     # изображение, чтобы цифры попадали на границу изображения
30     # сохраняются
31     изображение = cv2.imread (путь к изображению)
32     серый = cv2.cvtColor (изображение, cv2.COLOR_BGR2GRAY)
33     серый = cv2.copyMakeBorder (серый, 8, 8, 8, 8,
34                               cv2.BORDER_REPLICATE)

```

В строке 22 мы начинаем перебирать каждый из отдельных imagePath . Для каждого изображения мы загружаем его с диска (строка 31), преобразуйте его в оттенки серого (строка 32) и заполните границы изображения восемь пикселей в каждом направлении (строки 33 и 34). Рисунок 21.2 ниже показывает разницу между исходное изображение (слева) и дополненное изображение (справа).

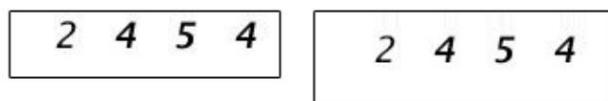


Рисунок 21.2: Слева: Исходное изображение, загруженное с диска. Справа: заполнение изображения, чтобы убедиться, что мы можем извлекать цифры только в том случае, если какая-либо из цифр касается границы изображения.

Мы выполняем это заполнение только в том случае, если какая-либо из наших цифр касается границы изображения. Если бы цифры касались границы, мы бы не смогли извлечь их из изображения. Таким образом, чтобы предотвратить эту ситуацию, мы намеренно дополняем входное изображение, чтобы данная цифра не могла коснуться границы.

Теперь мы готовы бинаризовать входное изображение с помощью метода пороговой обработки Оцу (глава 9, [Практический Python и OpenCV](#)):

```
36     # порог изображения, чтобы показать цифры
37     порог = cv2.threshold(серый, 0, 255,
38                           cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

Этот вызов функции автоматически ограничивает наше изображение таким образом, что наше изображение теперь двоичное — черное. Пиксели представляют собой фон, а белые пиксели — передний план, как показано на рис. 21.3.



Рисунок 21.3: Пороговое значение изображения гарантирует, что передний план будет белым, а фон черным.

Это типичное предположение/требование при работе со многими функциями обработки изображений с OpenCV.

Пороговое значение изображения является важным шагом в нашем конвейере обработки изображений, так как теперь нам нужно найти очертания каждой из цифр:

```
40     # найти контуры на изображении, оставив только четыре самых больших
41     # единицы
42     cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
43                             cv2.CHAIN_APPROX_SIMPLE)
44     cnts = cnts[0], если imutils.is_cv2() иначе cnts[1]
45     cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:4]
```

Строки 42 и 43 находят контуры (т.е. очертания) каждой из цифр на изображении. На всякий случай при наличии «шумов» на изображении сортируем контуры по их площади, оставляя только четыре самых больших (т.е. сами наши цифры).

Учитывая наши контуры, мы можем извлечь каждый из них, вычислив ограничивающую рамку:

```
47     # цикл по контурам
48     для с в центах:
```

```

49      # вычисляем ограничивающую рамку для контура, затем извлекаем
50      # цифра
51      (x, у, шир, выс) = cv2.boundingRect(c)
52      roi = серый [у - 5: у + выс + 5, x - шир: x + шир + 5]
53
54      # отображаем символ, делая его достаточно большим для нас
55      # чтобы увидеть, затем дождитесь нажатия клавиши
56      cv2.imshow ("ROI", imutils.resize (ROI, ширина = 28))
57      ключ = cv2.waitKey(0)

```

В строке 48 мы перебираем каждый из контуров, найденных в пороговом изображении. Мы называем cv2.boundingRect для вычисления координат ограничивающей рамки (x, у) цифровой области. Этот область интереса (ROI) затем извлекается из изображения в градациях серого в строке 52. Я включил пример цифр, извлеченных из необработанных изображений капчи в виде монтажа на рис. 21.4.

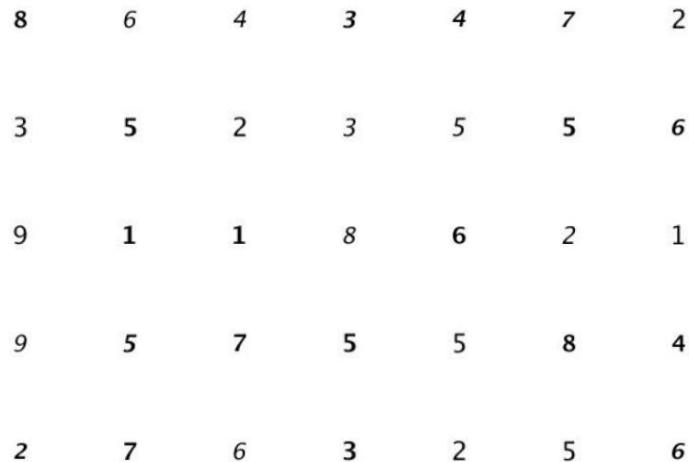


Рисунок 21.4: Образец цифровых ROI, извлеченных из наших изображений CAPTCHA. Нашей целью будет пометить эти изображения таким образом, чтобы мы могли обучить пользовательскую сверточную нейронную сеть на них.

Строка 56 отображает цифровой ROI на нашем экране, изменяя его размер, чтобы он был достаточно большим, чтобы мы могли легко видеть. Затем строка 57 ожидает нажатия клавиши на клавиатуре, но выбирайте ее с умом! Ключ вы нажмете будет использоваться в качестве метки для цифры.

Чтобы увидеть, как работает процесс маркировки с помощью вызова cv2.waitKey , взгляните на следующее. блок кода:

```

59      # если ""
60      # клавиша нажата, затем игнорируйте символ
61      если ключ == ord(""):
62          print("[INFO] игнорирует символ")
63          Продолжать
64
65      # взять нажатую клавишу и построить путь
66      # выходной каталог
67      ключ = chr(key).upper()
68      dirPath = os.path.sep.join([args["annot"], key])

```

```
69         # если выходной каталог не существует, создайте его
70     если нет os.path.exists(dirPath):
71         os.makedirs(путь_каталога)
```

Если нажата клавиша тильда ' (тильда), мы проигнорируем символ (строки 60 и 62). Необходимость может произойти игнорирование символа, если наш скрипт случайно обнаружит «шум» (т. е. что угодно, кроме цифры) во входном изображении или если мы не уверены, что это за цифра. В противном случае будем считать, что нажата клавиша был меткой для цифры (строка 66) и использовал ключ для создания пути к каталогу для нашего вывода метка (строка 67).

Например, если я нажму клавишу 7 на клавиатуре, dirPath будет:

набор данных/7

Поэтому все изображения, содержащие цифру «7», будут храниться в подкаталоге dataset/7 . Строки 70 и 71 проверяют, не существует ли директория dirPath — если это не так, мы создать это.

После того, как мы убедились, что dirPath правильно существует, нам просто нужно написать цифру примера в файл:

```
73             # записываем помеченный символ в файл
74         count = counts.get(ключ, 1)
75         p = os.path.sep.join([dirPath, "{}.png".format(
76             ул(количество).zfill(6))])
77         cv2.imwrite(p, roi)
78
79         # увеличить счетчик для текущего ключа
80         количество [ключ] = количество + 1
```

В строке 74 берется общее количество примеров, записанных на диск для текущей цифры. Мы затем создайте выходной путь к цифре примера, используя dirPath. После выполнения строк 75 и 76, наш выходной путь р может выглядеть так:

наборы данных/7/000001.png

Опять же, обратите внимание, как все примеры ROI, которые содержат число семь, будут сохранены в подкаталог datasets/7 — это простой и удобный способ организации наборов данных при маркировке картинки.

Наш последний блок кода обрабатывает, если мы хотим выйти из сценария, нажав Ctrl + C , или если есть ошибка обработка изображения:

```
82             # мы пытаемся вывести команду control-c из скрипта, так что оторвитесь от
83             # цикл (вам все еще нужно нажать клавишу для активного окна, чтобы
84             # вызвать это)
85         кроме KeyboardInterrupt:
86             print("[INFO] выход из скрипта вручную")
87             переменна
88
89         # для этого конкретного изображения произошла неизвестная ошибка
90         кроме:
91             print("[INFO] пропуск изображения...")
```

Если мы хотим нажать `ctrl+c` и выйти из скрипта досрочно, строка 85 обнаружит это и позволит нашей программе Python корректно завершить работу. Стока 90 перехватывает все остальные ошибки и просто игнорирует их, позволяя нам продолжить процесс маркировки.

Последнее, что вам нужно при маркировке набора данных, — это случайная ошибка из-за проблемы с кодировкой изображения, которая приведет к сбою всей вашей программы. Если это произойдет, вам придется перезапустить процесс маркировки заново. Очевидно, вы можете встроить дополнительную логику, чтобы определить, где вы остановились, но такой пример выходит за рамки этой книги.

Чтобы пометить изображения, загруженные с веб-сайта E-ZPass NY, просто выполните следующие действия. команда:

```
$ python annotate.py -- входные загрузки -- annot набор данных
```

Здесь вы можете видеть, что цифра 7 отображается на моем экране на рис. 21.5.



Рисунок 21.5: При аннотировании нашего набора данных цифр на экране будет отображаться заданная цифра ROI. Затем нам нужно нажать соответствующую клавишу на клавиатуре, чтобы пометить изображение и сохранить ROI на диск.

Затем я нажимаю клавишу 7 на клавиатуре, чтобы пометить ее, а затем цифра записывается в файл в подкаталоге `dataset/7`.

Затем сценарий `annotate.py` переходит к следующей цифре, которую я должен обозначить. Затем вы можете приступить к маркировке всех цифр в необработанных изображениях капчи. Вы быстро поймете, что маркировка набора данных может быть очень утомительным и трудоемким процессом. Нанесение меток на все 2000 цифр займет у вас менее получаса, но вам, скорее всего, станет скучно в течение первых пяти минут.

Помните, что на самом деле получить набор данных с метками — это полдела. Оттуда может начаться фактическая работа. К счастью, я уже обозначил для вас цифры! Если вы проверите каталог наборов данных, включенный в прилагаемые загрузки этой книги, вы обнаружите, что весь набор данных готов к работе:

```
$ ls набор данных/
1 2 3 4 5 6 7 8 9
$ ls -l набор данных/1/*.png | туалет -л
232
```

Здесь вы можете увидеть девять подкаталогов, по одному для каждой цифры, которую мы хотим распознать. Внутри каждого подкаталога есть примеры изображений конкретной цифры. Теперь, когда у нас есть помеченный набор данных, мы можем приступить к обучению нашего взломщика капчи с использованием архитектуры LeNet.

21.1.5 Предварительная обработка цифр Как

мы знаем, наши сверточные нейронные сети требуют передачи изображения фиксированной ширины и высоты во время обучения. Однако наши размеченные изображения цифр имеют разные размеры: некоторые в высоту больше, чем в ширину, другие в ширину больше, чем в высоту. Поэтому нам нужен метод для заполнения и изменения размера наших входных изображений до фиксированного размера без искажения их соотношения сторон.

Мы можем изменять размер и дополнять наши изображения, сохраняя соотношение сторон, определяя функцию предварительной обработки внутри `captcha helper.py`:

```

1 # импортируем необходимые пакеты
2 импорта imutils
3 импорт cv2
4
5 предварительная обработка по определению (изображение, ширина, высота):
6     # получаем размеры изображения, затем инициализируем
7     # значения заполнения
8     (ч, ш) = изображение.форма[:2]
9
10    # если ширина больше высоты, то изменить размер вдоль
11    # ширина
12    если ш > ч:
13        изображение = imutils.resize (изображение, ширина = ширина)
14
15    # в противном случае высота больше ширины, поэтому измените размер
16    # по высоте
17    еще:
18        изображение = imutils.resize (изображение, высота = высота)

```

Наша функция предварительной обработки требует три параметра:

1. изображение: входное изображение, которое мы собираемся дополнить и изменить размер.
2. ширина: целевая выходная ширина изображения.
3. высота: целевая высота вывода изображения.

В строках 12 и 13 мы проверяем, больше ли ширина высоты, и если да, то мы изменяем размер изображения по большему измерению (ширине) В противном случае, если высота больше, чем ширину мы изменяем по высоте (строки 17 и 18), что подразумевает либо ширину, либо высоту (в зависимости от размеров входного изображения) фиксированы.

Однако противоположное измерение меньше, чем должно быть. Чтобы решить эту проблему, мы можем «дополнить» изображение вдоль более короткого измерения, чтобы получить наш фиксированный размер:

```

20     # определить значения отступов для ширины и высоты
21     # получить целевые размеры
22     padW = int((ширина - image.shape[1]) / 2.0)
23     padH = int((высота - image.shape[0]) / 2.0)
24
25     # поместите изображение, затем примените еще одно изменение размера, чтобы справиться с любым
26     # проблемы с округлением
27     изображение = cv2.copyMakeBorder (изображение, padH, padH, padW, padW,
28                                         cv2.BORDER_REPLICATE)
29     изображение = cv2.resize (изображение, (ширина, высота))
30
31     # вернуть предварительно обработанное изображение
32     вернуть изображение

```

Строки 22 и 23 вычисляют необходимое количество отступов для достижения целевой ширины и высоты. Строки 27 и 28 применяют отступы к изображению. Применение этого заполнения должно привести к нашему изображению до нашей целевой ширины и высоты; однако могут быть случаи, когда мы ошибаемся на один пиксель в заданное измерение. Самый простой способ устранить это несоответствие — просто вызвать cv2.resize (Line 29), чтобы убедиться, что все изображения имеют одинаковую ширину и высоту.

Причина, по которой мы не вызываем сразу cv2.resize в начале функции, заключается в том, что мы сначала нужно рассмотреть соотношение сторон входного изображения и попытаться сначала правильно его заполнить. Если мы не соблюдайте соотношение сторон изображения, тогда наши цифры будут искажены.

21.1.6 Обучение анализатора капчи Теперь, когда наша

функция предварительной обработки определена, мы можем перейти к обучению LeNet на наборе данных изображения капчи. Откройте файл train_model.py и вставьте следующий код:

```

1 # импортируем необходимые пакеты 2 из
sklearn.preprocessing импортируем LabelBinarizer 3 из
sklearn.model_selection import train_test_split 4 из sklearn.metrics
importclassification_report 5 из keras.preprocessing.image import
img_to_array 6 из keras.optimizers import SGD 7 из pyimagesearch.nn.conv
импортировать LeNet 8 из pyimagesearch.utils.captchahelper импортировать
пропроцесс 9 из путей импорта imutils 10 импортировать matplotlib.pyplot
как plt 11 импортировать numpy как np 12 импортировать argparse 13
импортировать cv2 14 импортировать os

```

Строки 2-14 импортируют необходимые пакеты Python. Обратите внимание, что мы будем использовать оптимизатор SGD вместе с архитектурой LeNet для обучения модели цифрам. Мы также будем использовать нашу недавно определенную функцию предварительной обработки для каждой цифры, прежде чем передавать ее через нашу сеть.

Далее, давайте рассмотрим наши аргументы командной строки:

```

16 # построить разбор аргумента и разобрать аргументы 17 ap =
argparse.ArgumentParser() 18 ap.add_argument("-d", "-dataset", required=True,
19           help="путь к входному набору данных")
20 ap.add_argument("-m", "--model", required=True,
21           help="путь к выходной модели") 22
args = vars(ap.parse_args())

```

Сценарий train_model.py требует два аргумента командной строки: 1. --dataset: путь к входному набору данных помеченных цифр капчи (т. е. набор данных каталог на диске).
2. --model: Здесь мы указываем путь, по которому наши сериализованные веса LeNet будут сохранены после обучения.

Теперь мы можем загрузить наши данные и соответствующие метки с диска:

```

24 # инициализируем данные и метки
25 данных = [] 26 меток =
[] 27
28 # цикл по входным изображениям 29
для imagePath в paths.list_images(args["dataset"]): # загрузить изображение,
30     предварительно обработать его и сохранить в списке данных image =
cv2.imread(imagePath) image = cv2.cvtColor (изображение, cv2.COLOR_BGR2GRAY)
31     изображение = предварительная обработка (изображение, 28, 28) изображение =
32     img_to_array (изображение) data.append (изображение)
33
34
35

```

```

36
37     # извлечь метку класса из пути к изображению и обновить # список меток
38
39     label = imagePath.split(os.path.sep)[-2]
40     labels.append(метка)

```

В строках 25 и 26 мы инициализируем наши списки данных и меток соответственно. Затем мы перебираем каждое изображение в нашем наборе данных с меткой -data в строке 29. Для каждого изображения в наборе данных мы загружаем его с диска, преобразуем в оттенки серого и предварительно обрабатываем так, чтобы оно имело ширину 28 пикселей и высоту 28 пикселей (строки 31-35). Затем изображение преобразуется в массив, совместимый с Keras, и добавляется в список данных (строки 34 и 35).

Одно из основных преимуществ организации структуры каталогов набора данных в следующем формате:

корневой_каталог/метка_класса/имя_файла_изображения.jpg

состоит в том, что вы можете легко извлечь метку класса, взяв предпоследний компонент из имени файла (строка 39). Например, для входного пути dataset/7/000001.png метка будет 7, которая будет добавлена в список меток (строка 40).

Наш следующий блок кода обрабатывает нормализацию необработанных значений интенсивности пикселей в диапазоне [0,1], за которым следует путем построения тренировочных и тестовых сплитов, а также горячего кодирования меток:

```

42 # масштабируем необработанные интенсивности пикселей в
диапазоне [0, 1] 43 data = np.array(data, dtype="float") / 255.0 44 labels =
np.array(labels)
45
46 # разбить данные на обучающие и тестовые сплиты, используя 75% от 47 # данные
для обучения и оставшиеся 25% для тестирования 48 (trainX, testX, trainY, testY) =
train_test_split(data, labels, test_size=0.25, random_state= 42)
49
50
51 # преобразовать метки из целых чисел в векторы 52 lb =
LabelBinarizer().fit(trainY) 53 trainY = lb.transform(trainY) 54
testY = lb.transform(testY)

```

Затем мы можем инициализировать модель LeNet и оптимизатор SGD:

```

56 # инициализировать модель
57 print("[INFO] компиляция модели...") 58
model = LeNet.build(width=28, height=28, depth=1, class=9) 59 opt = SGD(lr=0.01) 60
model.compile (потеря = "categorical_crossentropy", оптимизатор = опция,
61               метрики=["точность"])

```

Наши входные изображения будут иметь ширину 28 пикселей, высоту 28 пикселей и один канал. Всего мы распознаем 9 классов цифр (нулевого класса нет).

Учитывая инициализированную модель и оптимизатор, мы можем обучить сеть на 15 эпох, оценить ее, и сериализовать его на диск:

```
63 # обучаем сеть
64 print("[INFO] обучающая сеть...")
65 H = model.fit(trainX, trainY, validation_data=(testX, testY),
66                 batch_size=32, epochs=15, verbose=1)
67
68 # оценить сеть
69 print("[INFO] оценка сети...") 70 прогнозы =
model.predict(testX, batch_size=32) 71 print(classification_report(testY.argmax(axis=1),
72
73
74 # сохранить модель на диск
75 print("[INFO] сериализующая сеть...") 76
model.save(args["model!"])
```

Наш последний блок кода будет отображать точность и потери как для обучающего, так и для тестового наборов с течением времени:

```
78 # график обучения + тестирование потерь и точности 79  
plt.style.use("ggplot") 80 plt.figure() 81 plt.plot(np.arange(0, 15),  
H.history["loss"], label="train_loss") 82 plt.plot(np.arange(0, 15),  
H.history["val_loss"], label="val_loss") 83 plt.plot(np.arange(0, 15), H.history["acc"], label="acc") 84  
plt.plot(np.arange(0, 15), H.history["val_acc"], label="val_acc") 85 plt.title("Обучение Потери и точность")  
86 plt.xlabel("Эпоха #") 87 plt.ylabel("Потери/точность") 88 plt.legend() 89 plt.show()
```

Чтобы обучить архитектуру LeNet с помощью оптимизатора SGD на нашем пользовательском наборе данных проверки, просто выполните следующую команду:

```
$ python train_model.py --dataset набор данных --model output/lenet.hdf5
[ИНФОРМАЦИЯ] модель компиляции...
[INFO] тренировочная сеть...
Обучение на 1509 образцах, проверка на 503 образцах Эпоха 1/15
0 с - потеря: 2,1606 - акк: 0,1895 - val_loss: 2,1553 - val_acc: 0,2266
Эпоха 2/15 0 с - потеря: 2,0877 - акк: 0,3565 - val_loss: 2,0874 - val_acc: 0,1769 Эпоха 3/15 0 с - потеря:
1,9540 - акк: 0,5003 - знач_потеря: 1,8878 - знач_акк: 0,3917
...
Эпоха 15/15 0 с
- потеря: 0,0152 - акк: 0,9993 - val_loss: 0,0261 - val_acc: 0,9980 [INFO] оценка сети...
```

точность	вспомнить	поддержку	f1-score
1	1,00	1,00	1,00
2	1,00	1,00	1,00

3	1,00	1,00	1,00	63
4	1,00	0,98	0,99	52
5	0,98	1,00	0,99	51
6	1,00	1,00	1,00	70
7	1,00	1,00	1,00	50
8	1,00	1,00	1,00	54
9	1,00	1,00	1,00	63
среднее / общее	1,00	1,00	1,00	503

[INFO] сериализация сети...

Как мы видим, всего через 15 эпох наша сеть достигает 100% точности классификации на как обучающий, так и проверочный наборы. Это также не случай переоснащения — когда мы исследуем кривых обучения и проверки на рис. 21.6 мы можем видеть, что к эпохе 5 проверка и проверка тренировочные потери/точность соответствуют друг другу.

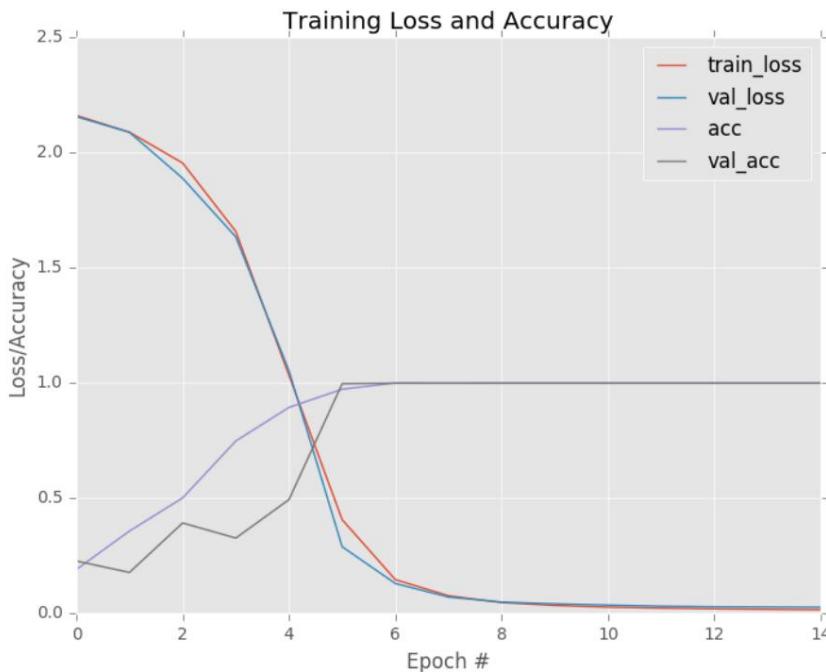


Рисунок 21.6: Использование архитектуры LeNet в наших пользовательских наборах данных цифр позволяет нам получить 100% точность классификации всего через пятнадцать эпох. Кроме того, отсутствуют признаки переобучения.

Если вы проверите выходной каталог, вы также увидите сериализованный файл lenet.hdf5:

```
$ ls -l вывод/
всего 9844
-rw-rw-r-- 1 адриан адриан 10076992 3 мая 12:56 lenet.hdf5
```

Затем мы можем использовать эту модель для новых входных изображений.

21.1.7 Тестирование разрушителя капчи

Теперь, когда наш взломщик капчи обучен, давайте проверим его на некоторых примерах изображений. Откройте файл `test_model.py` и вставьте следующий код:

```
1 # импортировать необходимые пакеты 2
из keras.preprocessing.image import img_to_array 3 из keras.models
import load_model 4 из pyimagesearch.utils.captchahelper import
preprocess 5 из imutils import контуры 6 из imutils import paths 7 import numpy
as np 8 import argparse 9 импорт imutils 10 импорт cv2
```

Как обычно, наш скрипт Python начинается с импорта наших пакетов Python. Мы снова будем использовать функция предварительной обработки для подготовки цифр к классификации.

Далее мы проанализируем наши аргументы командной строки:

```
12 # построить аргумент parse и разобрать аргументы 13 ap =
argparse.ArgumentParser() 14 ap.add_argument("-i", "--input", required=True,
help="путь к входному каталогу изображений") 16 ap.add_argument("-m", "--
15 model", required=True,
17 help="путь к входной модели") 18
args = vars(ap.parse_args())
```

Переключатель `-input` управляет путем к входным изображениям капчи, которые мы хотим взломать. Мы могли бы загрузить новый набор капч с веб-сайта E-ZPass NY, но для простоты мы возьмем образцы изображений из наших существующих необработанных файлов капчи. Аргумент `-model` — это просто путь к сериализованным весам, хранящимся на диске.

Теперь мы можем загрузить нашу предварительно обученную CNN и случайным образом выбрать десять изображений капчи для классификации:

```
20 # загрузить предварительно обученную сеть 21 print("[INFO]
загружает предварительно обученную сеть...") 22 model = load_model(args["model"])

23
24 # случайная выборка нескольких входных изображений
25 imagePaths = list(paths.list_images(args["input"])) 26 imagePaths =
np.random.choice(imagePaths, size=(10,), replace=False)
27
```

А вот и самое интересное — взлом капчи:

```
29 # цикл по путям изображения 30 для
imagePath в imagePaths: # загрузите
31     изображение и преобразуйте его в оттенки серого, затем дополните изображение
32     #, чтобы убедиться, что цифры, пойманые только на границе изображения, #
33     сохраняются
34     изображение = cv2.imread(путь к изображению)
```

```

35     серый = cv2.cvtColor(изображение, cv2.COLOR_BGR2GRAY)
36     серый = cv2.copyMakeBorder(серый, 20, 20, 20, 20,
37                               cv2.BORDER_REPLICATE)
38
39     # порог изображения, чтобы показать цифры
40     порог = cv2.threshold(серый, 0, 255,
41                           cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]

```

В строке 30 мы начинаем перебирать каждый из выбранных нами imagePath. Так же, как в annotate.py

Например, нам нужно извлечь каждую из цифр в капче. Это извлечение осуществляется путем загрузка изображения с диска, преобразование его в оттенки серого и дополнение границы таким образом, чтобы цифра не может касаться границы изображения (строки 34-37). Мы добавляем сюда дополнительные отступы, поэтому у нас есть достаточно места, чтобы на самом деле нарисовать и визуализировать правильный прогноз на изображении.

Строки 40 и 41 ограничивают изображение таким образом, чтобы цифры отображались как белый передний план на фоне черный фон.

Теперь нам нужно найти контуры цифр на изображении thresh:

```

43     # найти контуры на изображении, оставив только четыре самых больших,
44     # затем сортируем их слева направо
45     cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
46                             cv2.CHAIN_APPROX_SIMPLE)
47     cnts = cnts[0], если imutils.is_cv2() иначе cnts[1]
48     cnts = sorted(cnts, key=cv2.contourArea, reverse=True)[:4]
49     cnts = контуры.sort_contours(cnts)[0]
50
51     # инициализируем выходное изображение как изображение в оттенках серого с 3
52     # каналы вместе с выходными предсказаниями
53     вывод = cv2.merge([серый] * 3)
54     прогнозы = []

```

Мы можем найти цифры, вызвав cv2.findContours для изображения thresh . Эта функция возвращает список (x, y)-координат, определяющих контур каждой отдельной цифры.

Затем мы выполняем два этапа сортировки. На первом этапе контуры сортируются по размеру, сохраняя только самые большие четыре очертания. Мы (правильно) предполагаем, что четыре контура с наибольшим размером цифры, которые мы хотим распознать. Однако не существует гарантированного пространственного порядка, наложенного на эти контуры - третья цифра, которую мы хотим распознать, может быть первой в списке центров . Поскольку мы читаем цифры слева направо, нам нужно отсортировать контуры слева направо. Это достигается с помощью функция sort_contours (<http://pyimg.co/sbm9p>).

Строка 53 берет наше серое изображение и преобразует его в трехканальное изображение, реплицируя канал оттенков серого три раза (по одному для каждого красного, зеленого и синего каналов). Затем мы инициализируем наш список прогнозов CNN в строке 54.

Учитывая контуры цифр в капче, теперь мы можем ее разбить:

```

56     # цикл по контурам
57     для c в центах:
58         # вычисляем ограничивающую рамку для контура, затем извлекаем
59         # цифра
60         (x, y, w, h) = cv2.boundingRect(c)
61         roi = серый [y - 5: y + h + 5, x - 5: x + w + 5]
62
63         # предварительно обработать ROI и классифицировать его, а затем классифицировать

```

```

64         roi = предварительная обработка (roi, 28, 28)
65         roi = np.expand_dims (img_to_array (roi), ось = 0) / 255,0
66         pred = model.predict(roi).argmax(ось=1)[0] + 1
67         предсказания .append (ул (предыд.))
68
69         # рисуем предсказание на выходном изображении
70         cv2.rectangle (выход, (x - 2, y - 2),
71                         (x + ш + 4, y + ч + 4), (0, 255, 0), 1)
72         cv2.putText (выход, ул (пред), (x - 5, y - 5),
73                     cv2.FONT_HERSHEY_SIMPLEX, 0,55, (0, 255, 0), 2)

```

В строке 57 мы перебираем каждый из контуров (которые были отсортированы слева направо) цифры. Затем мы извлекаем ROI цифры в строках 60 и 61 с последующей предварительной обработкой в строках 60 и 61. Строки 64 и 65.

В строке 66 вызывается метод .predict нашей модели. Индекс с наибольшей вероятностью возвращаемый .predict будет нашей меткой класса. Мы добавляем 1 к этому значению, так как значения индексов начинаются на нуле; однако нулевого класса нет - только классы для цифр 1-9. Затем этот прогноз добавлен к списку прогнозов в строке 67.

Строки 70 и 71 рисуют ограничивающую рамку, окружающую текущую цифру, а строки 72 и 73 нарисуйте предсказанную цифру на самом выходном изображении.

Наш последний блок кода обрабатывает запись неработающей капчи в виде строки на наш терминал, а также отображение выходного изображения:

```

75     # показать выходное изображение
76     print("[INFO] captcha: {}".format("".join(предсказания)))
77     cv2.imshow ("Выход", вывод)
78     cv2.waitKey()

```

Чтобы увидеть наш взломщик капчи в действии, просто выполните следующую команду:

```

$ python test_model.py --входные загрузки --model output/lenet.hdf5
Использование бэкенда TensorFlow.
[INFO] загрузка предварительно обученной сети...
[ИНФО] капча: 2696
[ИНФО] капча: 2337
[ИНФО] капча: 2571
[ИНФО] капча: 8648

```

На рис. 21.7 я включил четыре образца, сгенерированные при запуске test_model.py. В каждом случае мы правильно предсказывали цифровую строку и разбивали капчу изображения с помощью простого сетевой архитектура, обученная на небольшом количестве обучающих данных.

21.2 Резюме

В этой главе мы узнали, как:

1. Соберите набор необработанных изображений.
2. Пометьте и аннотируйте наши изображения для обучения.
3. Обучите пользовательскую сверточную нейронную сеть на нашем размеченном наборе данных.
4. Протестируйте и оцените нашу модель на примерах изображений.

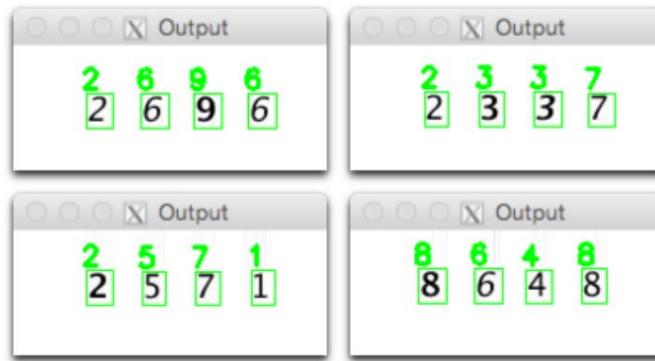


Рисунок 21.7: Примеры капч, которые были правильно классифицированы и взломаны нашей моделью LeNet .

Для этого мы взяли 500 примеров изображений капчи с веб-сайта E-ZPass NY.

Затем мы написали скрипт Python, который помогает нам в процессе маркировки, позволяя нам быстро маркировать весь набор данных и сохранять полученные изображения в организованной структуре каталогов.

После того, как наш набор данных был помечен, мы обучили архитектуру LeNet с помощью оптимизатора SGD на наборе данных с использованием категориальной кросс-энтропийной потери — полученная модель получила 100% точность на тестовом наборе с нулевым переоснащением. Наконец, мы визуализировали результаты предсказанных цифр, чтобы подтвердить , что мы успешно разработали метод взлома капчи.

Еще раз хочу напомнить вам, что эта глава служит только примером того, как получить набор данных изображения и пометить его. Ни при каких обстоятельствах вы не должны использовать этот набор данных или результирующую модель в гнусных целях. Если вы когда-нибудь окажетесь в ситуации, когда вы обнаружите, что компьютерное зрение или глубокое обучение могут быть использованы для использования уязвимости, обязательно практикуйте ответственное раскрытие информации и попытайтесь сообщить о проблеме соответствующим заинтересованным сторонам; невыполнение этого требования является неэтичным (как и неправильное использование этого кода, за которое с юридической точки зрения, я должен сказать, я не могу нести ответственность).

Во-вторых, в этой главе (как и в следующей, посвященной обнаружению улыбки с помощью глубокого обучения) использовалось компьютерное зрение и библиотека OpenCV для облегчения создания законченного приложения. Если вы планируете стать серьезным практиком глубокого обучения, я настоятельно рекомендую вам изучить основы обработки изображений и библиотеки OpenCV — даже элементарное понимание этих концепций позволит вам: 1. Оценить глубокое обучение на более высоком уровне..

2. Разрабатывайте более надежные приложения, использующие глубокое обучение для классификации изображений.
3. Используйте методы обработки изображений, чтобы быстрее достигать поставленных целей.

Отличный пример использования основных методов обработки изображений в наших интересах можно найти в разделе 21.1.4 выше, где мы смогли быстро аннотировать и пометить наш набор данных. Без использования простых методов компьютерного зрения нам бы пришлось вручную обрезать и сохранять примеры цифр на диск с помощью программного обеспечения для редактирования изображений, такого как Photoshop или GIMP. Вместо этого мы смогли написать быстрое и грязное приложение, которое автоматически извлекало каждую цифру из капчи — все, что нам нужно было сделать, это нажать соответствующую клавишу на клавиатуре, чтобы пометить изображение.

Если вы новичок в мире OpenCV или компьютерного зрения или просто хотите повысить уровень своих навыков, я настоятельно рекомендую вам поработать с моей книгой « [Практический Python и OpenCV](#) » [8]. Книга предназначена для быстрого чтения и даст вам основу, необходимую для успешного применения глубокого обучения к задачам классификации изображений и компьютерного зрения.

22. Практический пример: обнаружение улыбки

В этой главе мы создадим законченное комплексное приложение, которое сможет обнаруживать улыбки в видеопотоке в режиме реального времени, используя глубокое обучение наряду с традиционными методами компьютерного зрения.

Для выполнения этой задачи мы будем обучать архитектуру LeNet на наборе данных изображений, содержащих лица улыбающихся и не улыбающихся людей. Как только наша сеть будет обучена, мы создадим отдельный скрипт Python — он будет обнаруживать лица на изображениях с помощью встроенного в OpenCV каскадного детектора лиц Хаара, извлекать интересующую область лица (ROI) из изображения, а затем передавать ROI через LeNet для обнаружения улыбки.

При разработке реальных приложений для классификации изображений вам часто придется смешивать традиционные методы компьютерного зрения и обработки изображений с глубоким обучением. Я сделал все возможное, чтобы эта книга была самостоятельной с точки зрения алгоритмов, методов и библиотек, которые вам необходимо понять, чтобы добиться успеха при изучении и применении глубокого обучения. Однако полный обзор OpenCV и других методов компьютерного зрения выходит за рамки этой книги.

Чтобы быстро освоиться с OpenCV и основами обработки изображений, я рекомендую вам прочитать «[Практический Python и OpenCV](#)» — книга читается быстро, и на ее изучение у вас уйдет меньше выходных. К тому времени, когда вы закончите, у вас будет четкое понимание основ обработки изображений.

Для более глубокого изучения методов компьютерного зрения обязательно обратитесь к [курсу PyImage Search Gurus](#). Независимо от вашего опыта работы с компьютерным зрением и обработкой изображений, к тому времени, как вы закончите эту главу, у вас будет готовое решение для обнаружения улыбки, которое вы сможете использовать в своих собственных приложениях.

22.1 Набор данных SMILES

Набор данных SMILES состоит из изображений лиц, которые либо улыбаются, либо не улыбаются [51]. Всего в наборе данных 13 165 изображений в градациях серого, каждое из которых имеет размер 64× 64 пикселя.

Как показано на рис. 22.1, изображения в этом наборе данных сильно обрезаны вокруг лица, что облегчит процесс обучения, поскольку мы сможем изучать шаблоны «улыбается» или «не улыбается» непосредственно из входных изображений, точно так же, как мы делали в подобных главах ранее в этой книге.



Рисунок 22.1: Вверху: Примеры «улыбающихся» лиц. Внизу: образцы «не улыбающихся» лиц. В этой главе мы будем обучать сверточную нейронную сеть распознавать улыбающиеся и не улыбающиеся лица в видеопотоках в реальном времени.

Тем не менее, близкое кадрирование создает проблему во время тестирования — поскольку наши входные изображения будут содержать не только лицо, но и фон изображения, нам сначала нужно локализовать лицо на изображении и извлечь ROI лица, прежде чем мы сможем его передать через нашу сеть для обнаружения. К счастью, при использовании традиционных методов компьютерного зрения, таких как каскады Хаара, это намного проще, чем кажется.

Вторая проблема, которую нам нужно решить в наборе данных SMILES, — это дисбаланс классов. Хотя в наборе данных 13 165 изображений, 9 475 из этих примеров не улыбаются, и только 3 690 относятся к классу улыбающихся. Учитывая, что число «не улыбающихся» изображений более чем в 2,5 раза превышает количество «улыбающихся» примеров, мы должны быть осторожны при разработке нашей процедуры обучения.

Наша сеть может, естественно, выбрать ярлык «не улыбающееся», поскольку (1) распределения неравномерны и (2) в ней больше примеров того, как выглядит «не улыбающееся» лицо. Как мы увидим позже в этой главе, мы можем бороться с дисбалансом классов, вычисляя «вес» для каждого класса во время обучения.

22.2 Обучение улыбке CNN

Первым шагом в создании нашего детектора улыбки является обучение CNN на наборе данных SMILES различию улыбающегося и не улыбающегося лица. Чтобы выполнить эту задачу, давайте создадим новый файл с именем `train_model.py`. Оттуда вставьте следующий код:

```
1 # импортируем необходимые пакеты 2
из sklearn.preprocessing импортируем LabelEncoder 3 из
sklearn.model_selection import train_test_split 4 из sklearn.metrics
import classification_report 5 из keras.preprocessing.image import
img_to_array 6 из keras.utils import np_utils 7 из pyimagesearch.nn.conv
импортировать LeNet 8 из путей импорта imutils 9 импортировать
matplotlib.pyplot как plt 10 импортировать numpy как np 11
импортировать argparse 12 импортировать imutils 13 импортировать
cv2 14 импортировать os
```

Строки 2-14 импортируют необходимые пакеты Python. Мы использовали все пакеты раньше, но я хочу обратить ваше внимание на строку 7, куда мы импортируем класс LeNet (глава 14) — это Архитектура, которую мы будем использовать при создании нашего детектора улыбки.

Далее, давайте проанализируем наши аргументы командной строки:

```

16 # построить аргумент parse и разобрать аргументы
17 ap = argparse.ArgumentParser()
18 ap.add_argument("-d", "--dataset", required=True,
19                 help="путь к входному набору данных лиц")
20 ap.add_argument("-m", "--model", required=True,
21                 help="путь к выходной модели")
22 аргумента = переменные (ap.parse_args())
23
24 # инициализируем список данных и меток
25 данных = []
26 ярлыков = []

```

Нашему сценарию потребуются два аргумента командной строки, каждый из которых я подробно описал ниже:

1. --dataset: Путь к каталогу SMILES на диске.
2. --model: путь, по которому сериализованные веса LeNet будут сохранены после обучения.

Теперь мы готовы загрузить набор данных SMILES с диска и сохранить его в памяти:

```

28 # цикл по входным изображениям
29 для imagePath в отсортированном (список (paths.list_images (аргументы ["набор данных"]))):
30     # загрузить изображение, предварительно обработать его и сохранить в списке данных
31     изображение = cv2.imread (путь к изображению)
32     изображение = cv2.cvtColor (изображение, cv2.COLOR_BGR2GRAY)
33     изображение = imutils.resize (изображение, ширина = 28)
34     изображение = img_to_array (изображение)
35     data.append(изображение)
36
37     # извлечь метку класса из пути к изображению и обновить
38     # список ярлыков
39     label = imagePath.split(os.path.sep)[-3]
40     label = "улыбается", if label == "положительно", else "не улыбается"
41     labels.append(метка)

```

В строке 29 мы перебираем все изображения во входном каталоге --dataset . Для каждого из этих изображений мы:

1. Загрузите его с диска (строка 31).
2. Преобразуйте его в оттенки серого (строка 32).
3. Измените его размер, чтобы он имел фиксированный входной размер 28× 28 пикселей (строка 33).
4. Преобразуйте изображение в массив, совместимый с Keras и его порядком каналов (строка 34).
5. Добавьте изображение в список данных, на которых будет обучаться LeNet.

Строки 39-41 обрабатывают извлечение метки класса из imagePath и обновление списка меток .

Набор данных SMILES хранит улыбающиеся лица в подкаталоге SMILES/positives/positives7 .

в то время как не улыбающиеся лица живут в подкаталоге SMILES/negatives/negatives7 .

Следовательно, учитывая путь к изображению:

УЛЫБКИ/позитивы/позитивы7/10007.jpg

Мы можем извлечь метку класса, разделив разделитель пути к изображению и захватив предпоследний подкаталог: Positives. На самом деле это именно то, что делает Line 39 .

Теперь, когда наши данные и метки созданы, мы можем масштабировать необработанные интенсивности пикселей до диапазон [0,1], а затем применить однократное кодирование к меткам:

```
43 # масштабируем необработанные интенсивности пикселей в
диапазоне [0, 1] 44 data = np.array(data, dtype="float") / 255.0 45
labels = np.array(labels)
46
47 # преобразовать метки из целых чисел в векторы 48
le = LabelEncoder().fit(labels) 49 labels =
np_utils.to_categorical(le.transform(labels), 2)
```

Наш следующий блок кода решает проблему дисбаланса данных, вычисляя веса классов:

```
51 # учет перекоса в размеченных данных
52 classTotals = labels.sum(axis=0) 53 classWeight =
classTotals.max() / classTotals
```

Строка 52 вычисляет общее количество примеров на класс. В этом случае classTotals будет массив: [9475, 3690] для «не улыбающихся» и «улыбающихся» соответственно.

Затем мы масштабируем эти суммы в строке 53 , чтобы получить classWeight, используемый для обработки дисбаланса классов, что дает массив: [1, 2.56]. Это взвешивание означает, что наша сеть будет рассматривать каждый случай «улыбки» как 2,56 случая «не улыбаться» и помогает бороться с проблемой дисбаланса классов , увеличивая потери для каждого экземпляра на больший вес при просмотре «улыбающихся» примеров.

Теперь, когда мы вычислили наши веса классов, мы можем перейти к разбиению наших данных на обучение и тестирование разделены, используя 80% данных для обучения и 20% для тестирования:

```
55 # разбить данные на обучающие и тестовые сплиты, используя 80% из 56 # данных для обучения и оставшиеся 20%
для тестирования 57 (trainX, testX, trainY, testY) = train_test_split(data,
58
метки, test_size=0.2, stratify=метки, random_state=42)
```

Наконец, мы готовы обучать LeNet:

```
60 # инициализируем модель
61 print("[INFO] компиляция модели...")
model = LeNet.build(width=28, height=28, depth=1, class=2)
model.compile(loss="binary_crossentropy", оптимизатор= "Адам",
64     метрики=[ "точность"])
65
66 # обучаем сеть
67 print("[INFO] обучающая сеть...")
68 H = model.fit(trainX, trainY, validation_data=(testX, testY),
69     class_weight=classWeight, batch_size=64, epochs =15, verbose=1)
```

Строка 62 инициализирует архитектуру LeNet, которая будет принимать одноканальные изображения 28×28 . Учитывая, что есть только два класса (улыбается или не улыбается), мы устанавливаем class=2.

Мы также будем использовать `binary_crossentropy` вместо `categorical_crossentropy` в качестве функции потерь. Опять же, категориальная кросс-энтропия используется только тогда, когда количество классов больше двух.

До этого момента мы использовали оптимизатор SGD для обучения нашей сети. Здесь мы будем использовать Адама (строка 63) [113]. Я рассказываю о более продвинутых оптимизаторах (включая Adam, RMSprop, Adadelta) и других в пакете Practitioner Bundle; однако ради этого примера просто поймите, что Адам может сходиться быстрее, чем SGD в определенных ситуациях.

Опять же, оптимизатор и связанные с ним параметры часто считаются гиперпараметрами, которые необходимо настроить при обучении сети. Когда я собрал этот пример, я обнаружил, что Адам работает значительно лучше, чем SGD.

Линии 68 и 69 обучают LeNet в общей сложности 15 эпох, используя предоставленный `classWeight` для дисбаланса боевых классов.

Как только наша сеть обучена, мы можем оценить ее и сериализовать веса на диск:

```
71 # оценить сеть
72 print("[INFO] оценка сети...") 73 предсказания =
model.predict(testX, batch_size=64) 74 print(classification_report(testY.argmax(axis=1),
75     прогнозы.argmax(ось=1), target_names=le.classes_))
76
77 # сохранить модель на диск
78 print("[INFO] сериализующая сеть...") 79 model.save(args["model"])

```

Мы также построим кривую обучения для нашей сети, чтобы мы могли визуализировать производительность:

```
81 # график обучения + тестирование потерь и точности
82 plt.style.use("ggplot") 83 plt.figure() 84 plt.plot(np.arange(0,
15), H.history["loss"], label="train_loss") 85 plt.plot(np.arange(0,
15), H.history["val_loss"], label="val_loss") 86 plt.plot(np.arange(0, 15), H.history["acc"],
label="acc") 87 plt.plot(np.arange(0, 15), H.history["val_acc"], label="val_acc") 88
plt.title("Обучение Потери и точность") 89 plt.xlabel("Эпоха #") 90 plt.ylabel("Потери/
точность") 91 plt.legend() 92 plt.show()
```

Чтобы обучить наш детектор улыбки, выполните следующую команду:

```
$ python train_model.py -dataset ./datasets/SMILEsmileD \
output/lenet.hdf5 [INFO] компиляция модели...
```

[INFO] тренировочная сеть...
Обучение на 10532 образцах, проверка на 2633 образцах
Эпоха 1/15 8 с - потери: 0,3970 - акк: 0,8161 - val_loss:
0,2771 - val_acc: 0,8872 Эпоха 2/15 8 с - потери: 0,2572 - акк: 0,8919 - val_loss:
0,2620 - val_acc: 0,8899 Эпоха 3/15

```
7s - убыток: 0,2322 - акк: 0,9079 - val_loss: 0,2433 - val_acc: 0,9062
```

```
...
```

```
Эпоха 15/15
```

```
8s - убыток: 0,0791 - акк: 0,9716 - val_loss: 0,2148 - val_acc: 0,9351
```

```
[INFO] оценка сети...
```

	точность	вспомнить поддержку	f1-score	
не улыбается	0,95	0,97	0,96	1890 г.
улыбается	0,91	0,86	0,88	743
среднее / общее	0,93	0,94	0,93	2633

```
[INFO] сериализация сети...
```

После 15 эпох мы видим, что наша сеть достигает точности классификации 93%. Фигура 22.2 отображает нашу кривую обучения:

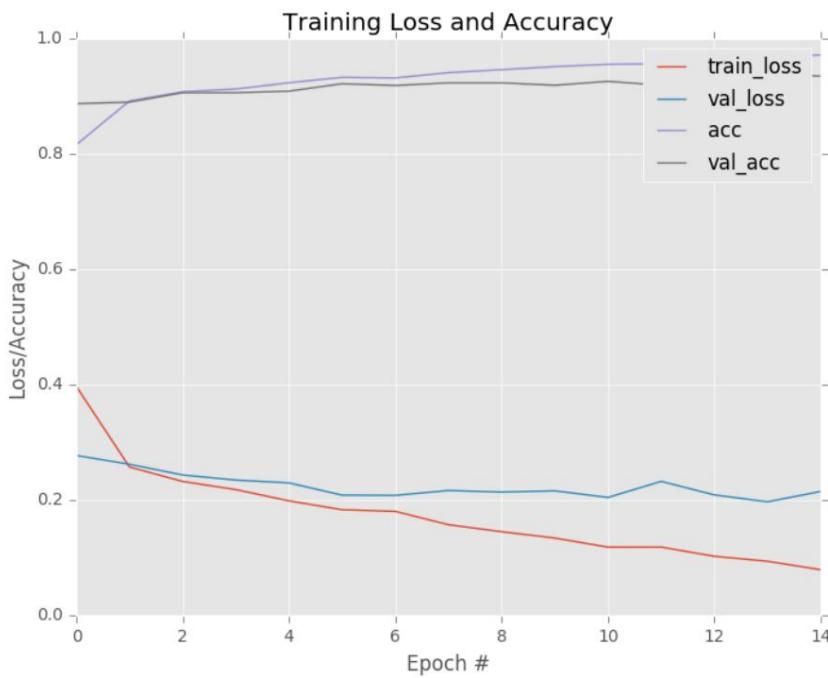


Рисунок 22.2: График кривой обучения для архитектуры LeNet, обученной на наборе данных SMILES. После пятнадцати эпох мы получаем точность классификации 93% на нашем тестовом наборе.

В прошлую шестую эпоху наша потеря валидации начинает стагнировать — результатом будет дальнейшее обучение после 15-й эпохи. В переоснащении. При желании мы могли бы повысить точность нашего детектора улыбки, используя больше тренировочные данные, либо:

1. Сбор дополнительных обучающих данных.
2. Применение увеличения данных для случайного перевода, поворота и смещения нашего существующего обучающего набора.

Расширение данных подробно описано в пакете Practitioner Bundle.

22.3 Запуск Smile CNN в режиме реального времени

Теперь, когда мы обучили нашу модель, следующим шагом будет создание скрипта Python для доступа к нашей веб-камере/видеофайлу и применения обнаружения улыбки к каждому кадру. Чтобы выполнить этот шаг, откройте новый файл, назовите его detect_smile.py, и мы приступим к работе.

```
1 # импортируем необходимые пакеты 2
из keras.preprocessing.image import img_to_array 3 из keras.models
import load_model 4 import numpy as np 5 import argparse 6 import
imutils 7 import cv2
```

Строки 2-7 импортируют необходимые пакеты Python. Функция img_to_array будет использоваться для преобразования каждого отдельного кадра из нашего видеопотока в правильно упорядоченный массив каналов. Функция load_model будет использоваться для загрузки весов нашей обученной модели LeNet с диска.

Для сценария Detectsmile.py требуются два аргумента командной строки, за которыми следует третий необязательный один:

```
9 # построить разбор аргумента и разобрать аргументы 10 ap =
argparse.ArgumentParser() 11 ap.add_argument("-c", "--cascade",
required=True,
12     help="путь к расположению каскада лиц")
13 ap.add_argument("-m", "--model", required=True,
14     help="путь к предварительно обученному детектору
улыбки CNN") 15 ap.add_argument("-v", "--video", help="путь к
16     (необязательному) видеофайлу")
17 аргументов = аргументы (ap.parse_args())
```

Первый аргумент --cascade — это путь к каскаду Хаара, используемому для обнаружения лиц на изображениях. Впервые опубликованный в 2001 году Полом Виолой и Майклом Джонсом каскад Хаара подробно описан в их работе «Быстрое обнаружение объектов с использованием усиленного каскада простых признаков» [134]. Эта публикация стала одной из самых цитируемых статей в литературе по компьютерному зрению.

Каскадный алгоритм Хаара способен обнаруживать объекты на изображениях независимо от их расположения и масштаба. Возможно, наиболее интригующим (и важным для нашего приложения) является то, что детектор может работать в режиме реального времени на современном оборудовании. На самом деле мотивация работы Виолы и Джонса заключалась в создании детектора лиц.

Поскольку подробный обзор обнаружения объектов с использованием традиционных методов компьютерного зрения выходит за рамки этой книги, вам следует просмотреть каскады Хаара, а также общую структуру гистограммы ориентированных градиентов + линейный SVM для обнаружения объектов, обратившись к этому сообщению в блоге PyImageSearch. (<http://pyimg.co/gq9lu>) вместе с модулем обнаружения объектов внутри PyImageSearch Gurus [33].

Второй аргумент общей строки, --model, указывает путь к нашим сериализованным весам LeNet на диске. Наш скрипт по умолчанию будет считывать кадры со встроенной/USB веб-камеры; однако, если вместо этого мы хотим читать кадры из файла, мы можем указать файл с помощью необязательного переключателя --video.

Прежде чем мы сможем обнаружить улыбки, нам сначала нужно выполнить некоторые инициализации:

```
19 # загрузить каскад детектора лица и детектор улыбки CNN
Детектор 20 = cv2.CascadeClassifier(аргументы["каскад"])
```

```

21 модель = load_model(аргументы["модель"])
22
23 # если видеотрек не указан, взять ссылку на веб-камеру
24 , если не args.get("видео", False):
25     камера = cv2.VideoCapture(0)
26
27 # иначе загрузить видео
28 еще:
29     камера = cv2.VideoCapture (аргументы ["видео"])

```

Строки 20 и 21 загружают каскадный детектор лиц Хаара и предварительно обученную модель LeNet соответственно. Если путь к видео не указан, мы берем указатель на нашу веб-камеру (строки 24 и 25).

В противном случае мы открываем указатель на видеофайл на диске (строки 28 и 29).

Теперь мы достигли основного конвейера обработки нашего приложения:

```

31 # продолжаем зацикливаться
32 , пока верно:
33     # захват текущего кадра
34     (снято, кадр) = камера.read()
35
36     # если мы просматриваем видео и мы не захватили кадр, то мы
37     # дошли до конца видео
38     если args.get("video") и не захвачен:
39         переменна
40
41     # изменить размер кадра, преобразовать его в оттенки серого, а затем клонировать
42     # исходный кадр, чтобы мы могли рисовать на нем позже в программе
43     кадр = imutils.resize (кадр, ширина = 300)
44     серый = cv2.cvtColor (кадр, cv2.COLOR_BGR2GRAY)
45     frameClone = кадр.копировать()

```

Строка 32 запускает цикл, который будет продолжаться до тех пор, пока (1) мы не остановим скрипт или (2) не достигнем конца видеофайла (при условии, что был применен путь --video).

Строка 34 захватывает следующий кадр из видеопотока. Если рамку не удалось захватить, то мы достигли конца видеофайла. В противном случае мы предварительно обрабатываем кадр для распознавания лиц, изменив его размер до ширины 300 пикселей (строка 43) и преобразовав его в оттенки серого (строка 44).

Метод .detectMultiScale обрабатывает определение координат ограничивающей рамки (x, y) лица в кадре:

```

47     # обнаруживать лица во входном кадре, затем клонировать кадр так, чтобы
48     # мы можем рисовать на нем
49     rects = детектор.detectMultiScale (серый, масштабный коэффициент = 1,1,
50             minNeighbors=5, minSize=(30, 30),
51             флаги = cv2.CASCADE_SCALE_IMAGE)

```

Здесь мы передаем наше изображение в градациях серого и указываем, что данная область считается лицом он должен иметь минимальную ширину 30× 30 пикселей. Атрибут minNeighbors помогает отсечь ложные срабатывания, в то время как scaleFactor управляет количеством пирамид изображений (<http://pyimg.co/rtped>) генерированные уровни.

Опять же, подробный обзор каскадов Хаара для обнаружения объектов выходит за рамки этой книги. Для более подробного ознакомления с распознаванием лиц в видеопотоках см. главу 15 [практического](#) руководства. [Python](#) и [OpenCV](#).

Метод `.detectMultiScale` возвращает список из 4 кортежей, составляющих прямоугольник, который ограничивает лицо в кадре. Первые два значения в этом списке являются начальными (x, y)-координатами.

Вторые два значения в списке прямоугольников — это ширина и высота ограничивающей рамки соответственно.

Мы перебираем каждый набор ограничивающих рамок ниже:

```

53      # цикл по ограничивающим прямоугольникам лица
54      для (fx, fy, fw, fh) в прямоугольниках:
55          # извлечь область интереса лица из изображения в градациях серого,
56          # измените его размер до фиксированных 28x28 пикселей, а затем подготовьте
57          # ROI для классификации через CNN
58          roi = серый[fY:fY + fh, fx:fx + fw]
59          roi = cv2.resize(roi, (28, 28))
60          roi = roi.astype ("с плавающей запятой") / 255,0
61          roi = img_to_array(roi)
62          roi = np.expand_dims (roi, ось = 0)

```

Для каждой из ограничивающих рамок мы используем нарезку массива NumPy, чтобы извлечь ROI лица (строка 58). Как только у нас есть ROI, мы предварительно обрабатываем его и подготавливаем к классификации через LeNet, изменяя его размер, масштабирование, преобразование в массив, совместимый с Keras, и добавление изображения с дополнительным измерением (строки 69-62).

После предварительной обработки области интереса ее можно передать через LeNet для классификации:

```

64      # определить вероятности как "улыбается", так и "не улыбается"
65      # улыбается", затем установите соответствующую метку
66      (не улыбается, улыбается) = model.predict(roi)[0]
67      label = "Улыбается", если улыбается > не улыбается, иначе "Не улыбается"

```

Вызов `.predict` в строке 66 возвращает вероятности «не улыбаться» и «улыбаться», соответственно. Стока 67 устанавливает метку в зависимости от того, какая вероятность больше.

Получив метку, мы можем нарисовать ее вместе с соответствующей ограничивающей рамкой на Рамка:

```

69      # отображаем метку и прямоугольник ограничивающей рамки на выходе
70      # Рамка
71      cv2.putText (frameClone, метка, (fx, fy - 10),
72                  cv2.FONT_HERSHEY_SIMPLEX, 0,45, (0, 0, 255), 2)
73      cv2.rectangle (frameClone, (fx, fy), (fx + fw, fy + fh),
74                  (0, 0, 255), 2)

```

Наш последний блок кода обрабатывает отображение выходного кадра на наш экран:

```

76      # показать наши обнаруженные лица вместе с метками улыбающихся/не улыбающихся
77      cv2.imshow ("Лицо", frameClone)
78
79      # если нажата клавиша 'q', останавливаем цикл
80      если cv2.waitKey(1) & 0xFF == ord("q"):
81          переменна
82
83      # очистить камеру и закрыть все открытые окна
84      камера.релиз()
85      cv2.destroyAllWindows()

```

Если нажата клавиша q, мы выходим из скрипта.

Чтобы запустить detect_smile.py с помощью вашей веб-камеры, выполните следующую команду:

```
$ python detect_smile.py --cascade haarcascade_frontalface_default.xml \ --model output/lenet.hdf5
```

Если вместо этого вы хотите использовать видеофайл (как я предоставил в сопроводительных загрузках этой книги), вы должны обновить свою команду, чтобы использовать переключатель --video:

```
$ python detect_smile.py --cascade haarcascade_frontalface_default.xml \ --model output/lenet.hdf5 --video path/to/your/video.mov
```

Я включил результаты скрипта обнаружения улыбки на рис. 22.3 ниже:

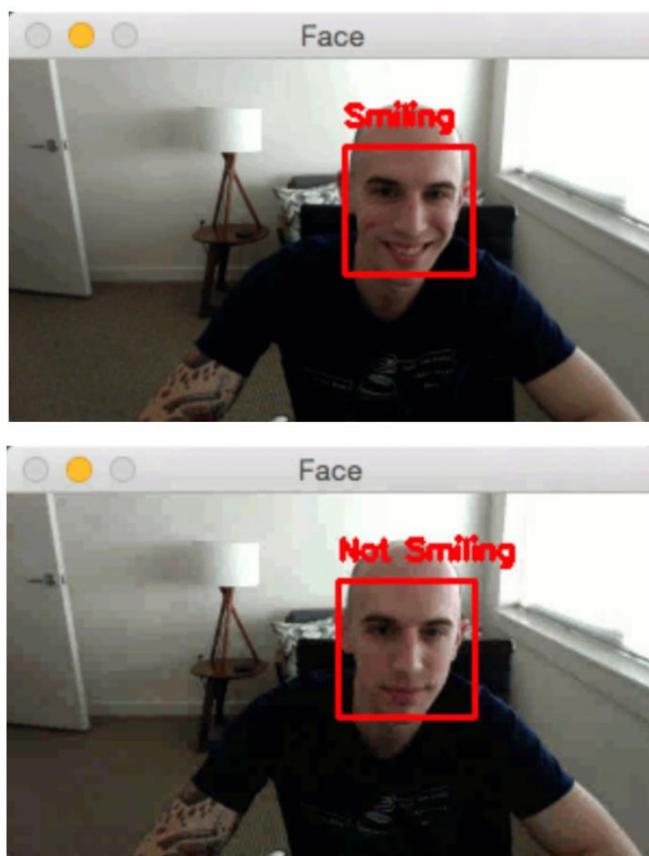


Рисунок 22.3: Применение нашей CNN для распознавания улыбки или отсутствия улыбки в видеопотоках в реальном времени на ЦП.

Обратите внимание, как LeNet правильно предсказывает «улыбается» или «не улыбается» на основе моего выражения лица.

22.4 Резюме

В этой главе мы узнали, как создать комплексное приложение компьютерного зрения и глубокого обучения для обнаружения улыбки. Для этого мы сначала обучили архитектуру LeNet на наборе данных SMILES.

Из-за дисбаланса классов в наборе данных SMILES мы обнаружили, как вычислять веса классов , чтобы помочь смягчить проблему.

После обучения мы оценили LeNet на нашем тестовом наборе и обнаружили, что сеть достигла приемлемой точности классификации 93% . Более высокую точность классификации можно получить, собрав больше обучающих данных или применив расширение данных к существующим обучающим данным.

Затем мы создали скрипт Python для чтения кадров с веб-камеры/видеофайла, обнаружения лиц и последующего применения нашей предварительно обученной сети. Для обнаружения лиц мы использовали каскады Хаара OpenCV. Как только лицо было обнаружено, оно извлекалось из кадра и затем передавалось через LeNet, чтобы определить, улыбается ли человек или нет. В целом, наша система обнаружения улыбки может легко работать в режиме реального времени на процессоре с использованием современного оборудования.

23. Ваши следующие шаги

Потратите секунду, чтобы поздравить себя; вы прошли весь начальный пакет Deep Learning for Computer Vision with Python. Это большое достижение, и вы его заслужили.

Давайте подумаем о вашем путешествии. В этой книге вы:

- Изучили основы классификации изображений. • Настроена среда глубокого обучения. • Создал свой первый классификатор изображений. • Изучал параметризованное обучение. • Узнал все об основных методах оптимизации (SGD) и методах регуляризации. • Изучил нейронные сети вдоль и поперек.
- Овладел основами сверточных нейронных сетей (CNN). • Обучил свой первый CNN. • Исследовал более продвинутые архитектуры, включая LeNet и MiniVGGNet.
- Научился определять недообучение и переоснащение. • Применение предварительно обученных CNN в наборе данных ImageNet для классификации ваших изображений. • Создал сквозную систему компьютерного зрения для взлома CAPTCHA. • Создал свой собственный детектор улыбки.

На данный момент у вас есть очень хорошее понимание основ машинного обучения, нейронных сетей и глубокого обучения, применяемых к компьютерному зрению. Но у меня такое чувство, что ваше путешествие только начинается...

23.1 Итак, что дальше?

Starter Bundle of Deep Learning for Computer Vision with Python — это лишь верхушка айсберга. Эта книга предназначена для того, чтобы помочь вам понять основы сверточных нейронных сетей, а также предоставить практические сквозные примеры / тематические исследования, которые вы можете использовать в качестве руководства при применении глубокого обучения к вашим собственным приложениям.

Но так же, как исследователи глубокого обучения обнаружили, что углубление приводит к более точным сетям, я также призываю вас глубже погрузиться в глубокое обучение.

Если хотите:

- Понимать более продвинутые методы обучения. •

Обучайте свои сети быстрее, используя трансферное обучение. • Работать с большими наборами данных, слишком большими для размещения в памяти. • Повысьте точность классификации с помощью сетевых ансамблей. • Изучите более экзотические архитектуры, такие как GoogLeNet и ResNet. • Изучите глубокие сновидения и нейронный стиль. • Узнайте о генеративно-состязательных сетях (GAN). • Обучайте современные архитектуры, такие как AlexNet, VGGNet, GoogLeNet, ResNet и SqueezeNet с нуля на сложном наборе данных ImageNet . . .

. . . то я настоятельно рекомендую вам не останавливаться на достигнутом. Продолжайте свой путь к мастерству глубокого обучения. Если вам понравился пакет Starter Bundle, я могу гарантировать, что с этого момента пакеты Practitioner Bundle и ImageNet Bundle станут только лучше.

Я надеюсь, что вы позволите мне продолжать направлять вас в вашем путешествии по глубокому обучению (и избегать тех же ошибок, что и я). Если вы еще не приобрели комплект Practitioner Bundle или ImageNet Bundle, вы можете сделать это здесь:

<https://www.pyimagesearch.com/deep-learning-computer-vision-python-book/> И если у вас есть какие-либо вопросы, не стесняйтесь обращаться ко мне: <http://www.pyimagesearch.com/contact/> Ура , – Адриан Роузброк

Список используемой литературы

- [1] Франсуа Шолле и др. Керас. <https://github.com/fchollet/keras>. 2015 (цитируется по стр. 18).
- [2] Тяньци Чен и др. «MXNet: гибкая и эффективная библиотека машинного обучения для гетерогенных распределенных систем» . В: arXiv.org (декабрь 2015 г.), arXiv: 1512.01274. arXiv: 1512.01274 [cs.DC] (цитируется на стр. 18).
- [3] Мартин Абади и др. TensorFlow: крупномасштабное машинное обучение в гетерогенных системах. Программное обеспечение доступно на сайте tensorflow.org. 2015. URL: <http://tensorflow.org/> (цитируется на стр. 18).
- [4] Команда разработчиков Theano. «Theano: платформа Python для быстрого вычисления математических выражений» . В: arXiv e-prints abs/1605.02688 (май 2016 г.). URL-адрес: <http://arxiv.org/abs/1605.02688> (цитируется на стр. 18).
- [5] Ф. Педрегоса и соавт. «Scikit-learn: машинное обучение в Python» . В: Journal of Machine Learning Research 12 (2011), страницы 2825–2830 (цитируется на страницах 19, 64).
- [6] Франсуа Шолле. Чем отличается Keras от других фреймворков глубокого обучения, таких как Tensor Flow, Theano или Torch? <https://www.quora.com/How-does-Keras-compare-to-other-Deep-Learning-frameworks-like-Tensor-Flow-Theano-or-Torch>. 2016 (цитируется на стр. 19).
- [7] Итsez. Библиотека компьютерного зрения с открытым исходным кодом (OpenCV). <https://github.com/itseez/opencv>. 2017 (цитируется на стр. 19).
- [8] Адриан Роузброк. Практические Python и OpenCV + тематические исследования. PyImageSearch.com, 2016. URL: <https://www.pyimagesearch.com/practical-python-opencv/> (цитируется на страницах 19, 38, 56, 306).
- [9] Ян Лекун, Йошуа Бенджио и Джейфри Хинтон. «Глубокое обучение» . In: Природа 521.7553 (2015), страницы 436–444 (цитируется на страницах 21, 126, 128).

- [10] Ян Гудфеллоу, Йошуа Бенджио и Аарон Курвиль. Глубокое обучение. <http://www. Deeplearningbook.org>. MIT Press, 2016 (цитируется на стр. 22, 24, 27, 42, 54, 56, 82, 95, 98, 113, 117, 169, 194, 252).
- [11] Уоррен С. МакКаллох и Уолтер Питтс. «Нейрокомпьютинг: основы исследований» . В: под редакцией Джеймса А. Андерсона и Эдварда Розенфельда. Кембридж, Массачусетс, США: MIT Press, 1988. Глава А Логическое исчисление идей, присущих нервной деятельности, страницы 15–27.
ISBN: 0-262-01097-6. URL: <http://dl.acm.org/citation.cfm?id=65669.104377> (цитируется на стр. 22).
- [12] Ф. Розенблatt. «Персептрон: вероятностная модель хранения и организации информации в мозгу» . В: Psychological Review (1958), страницы 65–386 (цитируется на страницах 22, 129, 130).
- [13] Ф. Розенблatt. Принципы нейродинамики: персептроны и теория механизмов мозга. Спартанец, 1962 г. (цит. по стр. 22).
- [14] М. Мински и С. Пейперт. Персептроны. Кембридж, Массачусетс: MIT Press, 1969 (цитируется на стр. 22, 129).
- [15] П.Дж. Вербос. «Помимо регрессии: новые инструменты прогнозирования и анализа в поведенческих науках» . Кандидатская диссертация. Гарвардский университет, 1974 г. (цитируется на стр. 23, 129).
- [16] Дэвид Э. Румельхарт, Джеффри Э. Хинтон и Рональд Дж. Уильямс. «Нейрокомпьютинг: основы исследований» . В: под редакцией Джеймса А. Андерсона и Эдварда Розенфельда. Кембридж, Массачусетс, США: MIT Press, 1988. Глава «Обучение представлений с помощью обратного распространения ошибок» , страницы 696–699. ISBN: 0-262-01097-6. URL-адрес: <http://dl.acm.org/citation.cfm?id=65669.104451> (цитируется на стр. 23, 129, 137).
- [17] Ян Лекун и др. «Эффективный BackProp» . В книге: Neural Networks: Tricks of the Trade эта книга является результатом семинара NIPS 1996 года. Лондон, Великобритания, Великобритания: Springer-Verlag, 1998, страницы 9–50. ISBN: 3-540-65311-2. URL: <http://dl.acm.org/citation.cfm?id=645754.668382> (цитируется на стр. 23, 166).
- [18] Балаш Чанад Чаджи. «Аппроксимация искусственными нейронными сетями» . В: Диссертация магистра наук, Университет Этвеша Лоранда (ELTE), Будапешт, Венгрия (2001 г.) (цитируется на стр. 23).
- [19] Янн Лекун и др. «Градиентное обучение для распознавания документов» . В: Продолжить IEEE. 1998, стр. 2278–2324 (цитируется на стр. 24, 195, 219, 227).
- [20] Джейсон Браунли. Что такое глубокое обучение? <http://machinelearningmastery.com/what-is-deep-learning/>. 2016 (цитируется на стр. 24).
- [21] Т. Ойала, М. Пиетикайнен и Т. Маенпaa. «Классификация текстур в оттенках серого с различным разрешением и инвариантная к вращению с локальными бинарными паттернами» . В: Pattern Analysis and Machine Intelligence, IEEE Transactions on 24.7 (2002), страницы 971–987 (цитируется на страницах 25, 51, 124).
- [22] Роберт М. Харалик, К. Шанмугам и Итс'Хак Динштейн. «Текстурные признаки для классификации изображений» . В: IEEE Transactions on Systems, Man, and Cybernetics SMC-3.6 (ноябрь 1973 г.), страницы 610–621. ISSN: 0018-9472. DOI: 10.1109/tsmc.1973.4309314. URL: <http://dx.doi.org/10.1109/tsmc.1973.4309314> (цитируется на стр. 25).
- [23] Мин-Куэй Ху. «Визуальное распознавание образов по моментным инвариантам» . В: Теория информации, IRE Transactions on 8.2 (февраль 1962 г.), страницы 179–187. ISSN: 0096-1000 (цитируется на стр. 25).
- [24] А. Хотанзад и Ю.Х. Хонг. «Распознавание инвариантных изображений с помощью моментов Зернике» . В: IEEE Trans. Аналитный узор. Max. Интел. 12.5 (май 1990 г.), страницы 489–497. ISSN: 0162-8828. DOI: 10.1109/34.55109. URL: <http://dx.doi.org/10.1109/34.55109> (цитируется на стр. 25).

- [25] Цзин Хуанг и др. «Индексирование изображений с использованием цветовых коррелограмм» . В: Материалы конференции 1997 г. по компьютерному зрению и распознаванию образов (CVPR '97). ЦВПР '97. Вашингтон, округ Колумбия, США: Компьютерное общество IEEE, 1997, стр. 762-. ISBN: 0-8186-7822-4. URL: <http://dl.acm.org/citation.cfm?id=794189.794514> (цитируется на стр. 25).
- [26] Эдвард Ростен и Том Драммонд. «Точки и линии слияния для высокопроизводительного отслеживания» . В: Труды Десятой международной конференции IEEE по компьютерному зрению - Том 2. ICCV '05. Вашингтон, округ Колумбия, США: Компьютерное общество IEEE, 2005 г., страницы 1508–1515. ISBN: 0-7695-2334-X-02. DOI: 10.1109/ICCV.2005.104. URL: <http://dx.doi.org/10.1109/ICCV.2005.104> (цитируется на стр. 25).
- [27] Крис Харрис и Майк Стивенс. «Комбинированный детектор углов и краев» . В: В проц. Четвертой конференции Alvey Vision. 1988, страницы 147–151 (цитируется на странице 25).
- [28] Дэвид Г. Лоу. «Распознавание объектов по локальным масштабно-инвариантным признакам» . В: Материалы Международной конференции по компьютерному зрению, том 2 - том 2. ICCV '99. Вашингтон, округ Колумбия, США: Компьютерное общество IEEE, 1999, стр. 1150-. ISBN: 0-7695-0164-8. URL: <http://dl.acm.org/citation.cfm?id=850924.851523> (цитируется на стр. 25).
- [29] Герберт Бэй и др. «Ускоренные надежные функции (SURF)» . В: Вычисл. Вис. Изображение Понимание. 110.3 (июнь 2008 г.), страницы 346–359. ISSN: 1077-3142. DOI: 10.1016/j.cviu.2007.09.014. URL: <http://dx.doi.org/10.1016/j.cviu.2007.09.014> (цитируется на стр. 25).
- [30] Майкл Калондер и соавт. «КОРОТКО: Двоичные надежные независимые элементарные функции» . В: Материалы 11-й Европейской конференции по компьютерному зрению: Часть IV. ECCB'10. Ираклион, Крит, Греция: Springer-Verlag, 2010, страницы 778–792. ISBN: 3-642-15560-X, 978-3-642-15560-4. URL: <http://dl.acm.org/citation.cfm?id=1888089.1888148> (цитируется на стр. 25).
- [31] Итан Рубли и др. «ORB: эффективная альтернатива SIFT или SURF» . В: Материалы Международной конференции по компьютерному зрению 2011 года. MKB '11. Вашингтон, округ Колумбия, США: Компьютерное общество IEEE, 2011 г., страницы 2564–2571. ISBN: 978-1-4577-1101-5. DOI: 10.1109/ICCV.2011.6126544. URL: <http://dx.doi.org/10.1109/ICCV.2011.6126544> (цитируется на стр. 25).
- [32] Навнит Далал и Билл Триггс. «Гистограммы ориентированных градиентов для обнаружения человека» . В: Материалы конференции IEEE Computer Society 2005 г. по компьютерному зрению и распознаванию образов (CVPR'05) - Том 1 - Том 01. CVPR '05. Вашингтон, округ Колумбия, США: Компьютерное общество IEEE, 2005 г., страницы 886–893. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.177. URL: <http://dx.doi.org/10.1109/CVPR.2005.177> (цитируется на стр. 25, 51, 124).
- [33] Адриан Роузброк. Гуру PyImageSearch. <https://www.pyimagesearch.com/pyimagesearch> гуру/. 2016 (цитируется на стр. 26, 27, 34, 38, 75, 313).
- [34] Педро Ф. Фельценшвалб и др. «Обнаружение объектов с дискриминационно обученными моделями на основе частей» . В: IEEE Trans. Аналитный узор. Max. Интел. 32.9 (сентябрь 2010 г.), страницы 1627–1645. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2009.167. URL-адрес: <http://dx.doi.org/10.1109/TPAMI.2009.167> (цитируется на стр. 26).
- [35] Томаш Малисевич, Абхинав Гупта и Алексей А. Эфрос. «Ансамбль Exemplar-SVM для обнаружения объектов и не только» . В: ICCV. 2011 (цитируется на стр. 26).
- [36] Джек Дин. Результаты улучшаются благодаря большему объему данных, более крупным моделям, большему количеству вычислений. <http://static.googleusercontent.com/media/research.google.com/en/people/jeff/BayLearn2015.pdf>. 2016 (цитируется на стр. 27).

- [37] Джеффри Хинтон. Что на самом деле было не так с обратным распространением в 1986 году? https://www.youtube.com/watch?v=VhmE_UXDOGs. 2016 (цитируется на стр. 28).
- [38] Эндрю Нг. Глубокое обучение, самообучение и неконтролируемое функциональное обучение. <https://www.youtube.com/watch?v=n1ViNeWhC24>. 2013 г. (цитируется на стр. 29).
- [39] Эндрю Нг. Что специалисты по данным должны знать о глубоком обучении. <https://www.slideshare.net/ExtractConf>. 2015 (цитируется на стр. 29).
- [40] Юрген Шмидхубер. «Глубокое обучение в нейронных сетях: обзор» . В: CoRR abs/1404.7828 (2014). URL: <http://arxiv.org/abs/1404.7828> (цитируется на стр. 29, 128).
- [41] Сатья Маллик. Почему OpenCV использует цветовой формат BGR? <http://www.learnopencv.com/why-does-opencv-use-bgr-color-format/>. 2015 (цитируется на стр. 36).
- [42] Ольга Русаковская и др. «Масштабная задача визуального распознавания ImageNet» . В: Международный журнал компьютерного зрения (IJCV) 115.3 (2015), страницы 211–252. DOI: 10.1007/s11263-015-0816-y (цитируется на стр. 45, 58, 68, 81, 277).
- [43] Коринна Кортес и Владимир Вапник. «Сети опорных векторов» . В: Max. Учиться. 20.3 (сентябрь 1995 г.), страницы 273–297. ISSN: 0885-6125. DOI: 10.1023/A:1022627411411. URL: <http://dx.doi.org/10.1023/A:1022627411411> (цитируется на стр. 45, 89).
- [44] Бернхард Э. Бозер, Изабель М. Гийон и Владимир Н. Вапник. «Алгоритм обучения для оптимальных классификаторов маржи» . В: Труды пятого ежегодного семинара по вычислительной теории обучения. КОЛТЬ 92 года. Питтсбург, Пенсильвания, США: ACM, 1992, страницы 144–152. ISBN: 0-89791-497-X. DOI: 10.1145/130385.130401. URL: <http://doi.acm.org/10.1145/130385.130401> (цитируется на стр. 45).
- [45] Лео Брейман. «Случайные леса» . В: Max. Учиться. 45.1 (октябрь 2001 г.), страницы 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: <http://dx.doi.org/10.1023/A:1010933404324> (цитируется на стр. 45).
- [46] Денни Чжоу и др. «Обучение с локальной и глобальной согласованностью» . В: Достижения в системах обработки нейронной информации 16. Под редакцией С. Труна, Л. К. Саула и П. Б. Шелькопфа. MIT Press, 2004, страницы 321–328. URL: <http://papers.кусает.cc/paper/2506-Learning-with-local-and-global-consistency.pdf> (цитируется на стр. 48).
- [47] Сюджин Чжу и Зубин Гахрамани. Обучение на размеченных и неразмеченных данных с помощью Распространение этикетки. Технический отчет. 2002 г. (цитируется на стр. 48).
- [48] Антти Расмус и др. «Полуконтролируемое обучение с лестничной сетью» . В: CoRR abs/1507.02672 (2015). URL: <http://arxiv.org/abs/1507.02672> (цитируется на стр. 48).
- [49] Аврим Блюм и Том Митчелл. «Объединение размеченных и неразмеченных данных с совместным обучением» . В: Труды одиннадцатой ежегодной конференции по вычислительной теории обучения. COLT' 98. Мэдисон, Висконсин, США: ACM, 1998, стр. 92–100. ISBN: 1-58113-057-0. DOI: 10.1145/279943.279962. URL: <http://doi.acm.org/10.1145/279943.279962> (цитируется на стр. 48).
- [50] Алекс Крижевский, Винод Наир и Джеки Хинтон. CIFAR-10 и CIFAR-100 (Канадский институт перспективных исследований). <http://www.cs.toronto.edu/~kriz/cifar.html> (цитируется на стр. 55).
- [51] Даниил Громада. SMILEулыбаетсяD. <https://github.com/hromi/SMILEsmileD>. 2010 (цитируется на стр. 55, 307).

- [52] Мария-Елена Нильсбак и Эндрю Зиссерман. «Визуальный словарь для классификации цветов» . В: ЦВПР (2). Компьютерное общество IEEE, 2006 г., страницы 1447–1454. URL : <http://dblp.uni-trier.de/db/conf/cvpr/cvpr2006-2.html#NilsbackZ06> (цитируется на стр. 56).
- [53] Л. Фей-Фей, Р. Фергус и Пьетро Перона. «Изучение генеративных визуальных моделей на нескольких обучающих примерах: поэтапный байесовский подход, протестированный на 101 категории объектов» . В: 2004 (цитируется на стр. 57).
- [54] Кристен Грауман и Тревор Даррелл. «Ядро соответствия пирамиды: эффективное обучение с наборами функций» . В: Дж. Max. Учиться. Рез. 8 (май 2007 г.), страницы 725–760. ISSN: 1532-4435. URL: <http://dl.acm.org/citation.cfm?id=1248659.1248685> (цитируется на стр. 57).
- [55] Светлана Лазебник, Корделия Шмид и Жан Понсе. «Помимо набора функций: сопоставление пространственных пирамид для распознавания категорий естественных сцен» . В: Материалы конференции IEEE Computer Society 2006 г. по компьютерному зрению и распознаванию образов - Том 2. CVPR '06. Вашингтон, округ Колумбия, США: Компьютерное общество IEEE, 2006 г., страницы 2169–2178. ISBN: 0-7695-2597-0. DOI: 10.1109/CVPR.2006.68. URL-адрес: <http://dx.doi.org/10.1109/CVPR.2006.68> (цитируется на стр. 57).
- [56] Хао Чжан и др. «SVM-KNN: дискrimинационная классификация ближайших соседей для визуального распознавания категорий» . В: Материалы конференции IEEE Computer Society 2006 г. по компьютерному зрению и распознаванию образов - Том 2. CVPR '06. Вашингтон, округ Колумбия, США: Компьютерное общество IEEE, 2006 г., страницы 2126–2136. ISBN: 0-7695-2597-0. DOI: 10.1109/CVPR.2006.301. URL: <http://dx.doi.org/10.1109/CVPR.2006.301> (цитируется на стр. 57).
- [57] Андрей Карпаты. CS231n: сверточные нейронные сети для визуального распознавания. <http://cs231n.stanford.edu/>. 2016 (цитируется на стр. 57, 84, 94, 106, 137, 142).
- [58] Эран Эйдингер, Рои Энбар и Тал Хасснер. «Возрастная и гендерная оценка нефильтрованных лиц» . В: Пер. Информация. За. сек. 9.12 (декабрь 2014 г.), страницы 2170–2179. ISSN: 1556-6013. DOI: 10.1109/TIFS.2014.2359646. URL-адрес: <http://dx.doi.org/10.1109/TIFS.2014.2359646> (цитируется на стр. 58).
- [59] Ворднет. О Ворднет. <http://wordnet.princeton.edu>. 2010 (цитируется на стр. 58).
- [60] А. Куаттони и А. Торральба. «Распознавание внутренних сцен» . В: Компьютерное зрение и распознавание образов, Конференция компьютерного общества IEEE. Лос-Аламитос, Калифорния, США: Компьютерное общество IEEE, 2009 г., стр. 413–420 (цитируется на стр. 60).
- [61] Джонатан Краузе и др. «Трехмерные представления объектов для мелкозернистой категоризации» . В: 4-й международный семинар IEEE по трехмерному представлению и распознаванию (3dRR-13). Сидней, Австралия, 2013 г. (цитируется на стр. 60).
- [62] Стефан ван дер Вальт и др. «scikit-image: обработка изображений в Python» . В: PeerJ 2 (июнь 2014 г.), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: <http://dx.doi.org/10.7717/peerj.453> (цитируется на стр. 64).
- [63] Майк Гручи. Будьте Pythonic: __init__.py. http://mikegrouchy.com/blog/2012/05/be-pythonic-init_py.html. 2012 г. (цитируется на стр. 69).
- [64] Ольга Векслер. к Ближайшие соседи. http://www.csd.uwo.ca/courses/CS9840a/Lekcija2_knn.pdf. 2015 (цитируется на стр. 72).
- [65] Джон Луи Бентли. «Многомерные двоичные деревья поиска, используемые для ассоциативного поиска» . В: Общ. ACM 18.9 (сентябрь 1975 г.), страницы 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: <http://doi.acm.org/10.1145/361002.361007> (цитируется на стр. 79).

- [66] Мариус Муджа и Дэвид Г. Лоу. «Масштабируемые алгоритмы ближайшего соседа для данных высокой размерности» . В: Pattern Analysis and Machine Intelligence, IEEE Transactions on 36 (2014) (цитируется на страницах 79, 81).
- [67] Санджой Дасгупта. «Эксперименты со случайной проекцией» . В: Материалы 16-й конференции по неопределенности в искусственном интеллекте. УАИ '00. Сан-Франциско, Калифорния, США: Morgan Kaufmann Publishers Inc., 2000, страницы 143–151. ISBN: 1-55860-709-9. URL: <http://dl.acm.org/citation.cfm?id=647234.719759> (цитируется на стр. 79).
- [68] Элла Бингем и Хейкки Маннила. «Случайная проекция при уменьшении размерности: приложения к изображениям и текстовым данным» . В: Материалы седьмой Международной конференции ACM SIGKDD по обнаружению знаний и интеллектуальному анализу данных. КДД '01. Сан-Франциско, Калифорния: ACM, 2001, страницы 245–250. ISBN: 1-58113-391-X. DOI: 10.1145/502512.502546. URL: <http://doi.acm.org/10.1145/502512.502546> (цитируется на стр. 79).
- [69] Санджой Дасгупта и Анупам Гупта. «Элементарное доказательство теоремы Джонсона и Линденштрауса» . В: Случайная структура. Алгоритмы 22.1 (январь 2003 г.), страницы 60–65. ISSN: 1042-9832 . DOI: 10.1002/rsa.10073. URL: <http://dx.doi.org/10.1002/rsa.10073> (цитируется на стр. 79).
- [70] Педро Домингос. «Несколько полезных вещей, которые нужно знать о машинном обучении» . В: Общ. ACM 55.10 (октябрь 2012 г.), страницы 78–87. ISSN: 0001-0782. DOI: 10.1145/2347736.2347755. URL: <http://doi.acm.org/10.1145/2347736.2347755> (цитируется на стр. 79, 80).
- [71] Дэвид Маунт и Сунил Айра. ANN: библиотека для приблизительного поиска ближайших соседей. <https://www.cs.umd.edu/~mount/ANN/>. 2010 (цитируется на стр. 81).
- [72] Эрик Бернхардссон. Раздражает: приблизительные ближайшие соседи в C++/Python, оптимизированные для использования памяти и загрузки/сохранения на диск. <https://github.com/spotify/annoy>. 2015 (цитируется на стр. 81).
- [73] Стюарт Рассел и Питер Норвиг. Искусственный интеллект: современный подход. 3-й. Река Аппер - Сэдл, Нью-Джерси, США: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594 (цитируется на стр. 82).
- [74] Андрей Карпаты. Линейная классификация. <http://cs231n.github.io/linear-classify/> (цитируется на стр. 82, 94).
- [75] П.Н. Клейн. Кодирование матрицы: линейная алгебра через приложения к информатике. Newtonian Press, 2013. ISBN: 9780615880990. URL: <https://books.google.com/books?id=3AA4nwEACAAJ> (цитируется на стр. 84).
- [76] Эндрю Нг. Машинное обучение. <https://www.coursera.org/learn/machine-learning> (цитируется на стр. 88, 132, 137, 141).
- [77] Ян Х. Виттен, Эйбе Франк и Марк А. Холл. Интеллектуальный анализ данных: практические инструменты и методы машинного обучения. 3-й. Сан-Франциско, Калифорния, США: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123748569, 9780123748560 (цитируется на стр. 88).
- [78] Питер Харрингтон. Машинное обучение в действии. Гринвич, Коннектикут, США: Manning Publications Co., 2012. ISBN: 1617290181, 9781617290183 (цитируется на стр. 88).
- [79] Стивен Марсленд. Машинное обучение: алгоритмическая перспектива. 1-й. Chapman & Hall/CRC, 2009. ISBN: 1420067184, 9781420067187 (цитируется на стр. 88).
- [80] Михаил Зибулевский. Домашнее задание по аналитическому и численному вычислению градиента и гессиана. <https://www.youtube.com/watch?v=ruuW4-InUxM> (цитируется на стр. 98).

- [81] Эндрю Нг. Конспект лекций CS229. <http://cs229.stanford.edu/lectures/cs229-notes1.pdf> (цитируется на стр. 98).
- [82] Андрей Карпаты. Оптимизация. <http://cs231n.github.io/optimization-1/> (цитируется на странице 98).
- [83] Андрей Карпаты. Лекция 3: Функции потерь и оптимизация. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture3.pdf (цитируется на стр. 99).
- [84] Дмитрий Мишкин и Иржи Матас. «Все, что вам нужно, это хорошее начало». В: CoRR abs/1511.06422 (2015). URL: <http://arxiv.org/abs/1511.06422> (цитируется на стр. 102).
- [85] Стэнфордский университет. Стэнфордские лаборатории электроники и др. Адаптивный «адалиновый» нейрон с использованием химических «мемисторов». 1960. URL: <https://books.google.com/books?id=Yc4EAAAAIAAJ> (цитируется на стр. 106).
- [86] Нин Цянь. «О термине импульса в алгоритмах обучения градиентного спуска». В: Нейронная сеть. 12.1 (январь 1999 г.), страницы 145–151. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(98)00116-6. URL: [http://dx.doi.org/10.1016/S0893-6080\(98\)00116-6](http://dx.doi.org/10.1016/S0893-6080(98)00116-6) (цитируется на стр. 111, 112).
- [87] Юрий Нестеров. «Метод решения задачи выпуклого программирования со скоростью сходимости $O(1/k^2)$ ». В: Доклады советской математики. Том 27. 2. 1983 г., страницы 372–376 (цитируется на страницах 111, 112).
- [88] Себастьян Рудер. «Обзор алгоритмов оптимизации градиентного спуска». В: CoRR abs/1609.04747 (2016). URL: <http://arxiv.org/abs/1609.04747> (цитируется на стр. 112).
- [89] Ричард С. Саттон. «Две проблемы с обратным распространением и другими процедурами обучения наискорейшего спуска для сетей». В: Материалы восьмой ежегодной конференции Общества когнитивных наук. Хиллсдейл, Нью-Джерси: Erlbaum, 1986 (цитируется на стр. 112).
- [90] Джон Хинтон. Нейронные сети для машинного обучения. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (цитируется на стр. 112, 164).
- [91] Йошуа Бенджио, Никола Буланже-Левандовски и Разван Паскану. «Достижения в оптимизации рекуррентных сетей». В: CoRR abs/1212.0901 (2012). URL-адрес: <http://arxiv.org/abs/1212.0901> (цитируется на стр. 112).
- [92] Илья Суцкевер. «Обучение рекуррентных нейронных сетей». ААИНС22066. Кандидатская диссертация. Торонто, Онтарио, Канада, 2013 г. ISBN: 978-0-499-22066-0 (цитируется на стр. 112).
- [93] Андрей Карпаты. Нейронные сети (часть III). http://cs231n.github.io/neural_network-3/ (цитируется на стр. 113, 253).
- [94] Алекс Крижевский, Илья Суцкевер и Джон Хинтон. «Классификация ImageNet с глубокими свёрточными нейронными сетями». В: Достижения в системах обработки нейронной информации 25. Под редакцией F. Pereira et al. Curran Associates, Inc., 2012 г., страницы 1097–1105. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf> (цитируется на стр. 113, 185, 192, 229).
- [95] Карен Симонян и Эндрю Зиссерман. «Очень глубокие сверточные сети для крупномасштабного распознавания изображений». В: CoRR abs/1409.1556 (2014). URL: <http://arxiv.org/abs/1409.1556> (цитируется на стр. 113, 192, 195, 227, 229, 278).
- [96] Kaiming He et al. «Глубокое остаточное обучение для распознавания изображений». В: CoRR abs/1512.03385 (2015). URL: <http://arxiv.org/abs/1512.03385> (цитируется на стр. 113, 126, 188, 192, 279).

- [97] Кристиан Сегеди и соавт. «Погружаемся глубже с извилинами» . В: Компьютерное зрение и распознавание образов (CVPR). 2015. URL: <http://arxiv.org/abs/1409.4842> (цитируется на стр. 113, 192, 280).
- [98] Хуэй Цзоу и Тревор Хасти. «Регуляризация и выбор переменных через эластичную сеть» . В: Журнал Королевского статистического общества, серия B 67 (2005 г.), страницы 301–320 (цитируется на страницах 113, 116).
- [99] Участники DeepLearning.net. Документация по глубокому обучению: регуляризация. <http://deeplearning.net/tutorial/gettingstarted.html#regularization> (цитируется на стр. 117).
- [100] Андрей Карпаты. Нейронные сети (Часть II). [http://cs231n.github.io/neural network-2/](http://cs231n.github.io/neural_network-2/) (цитируется на стр. 117).
- [101] Richard HR Hahnloser et al. «Цифровая селекция и аналоговое усиление существуют в кремниевой схеме, вдохновленной кортексом» . В: Nature 405.6789 (2000), стр. 947 (цитируется на стр. 126, 128).
- [102] Ричард Х.Р. Ханлозер, Х. Себастьян Сеунг и Жан-Жак Слотин. «Разрешенные и запрещенные множества в симметричных пороговых линейных сетях» . В: Нейронные вычисления. 15.3 (март 2003 г.), страницы 621–638. ISSN: 0899-7667. DOI: 10.1162/089976603321192103. URL: <http://dx.doi.org/10.1162/089976603321192103> (цитируется на стр. 126).
- [103] Эндрю Л. Маас, Авни Ю. Ханнун и Эндрю Ю. Нг. «Нелинейности выпрямителя улучшают акустические модели нейронной сети» . In: in ICML Workshop on Deep Learning for Audio, Speech and Language Processing. 2013 (цитируется на стр. 126).
- [104] Джорк-Арне Клеверт, Томас Унтертинер и Зепп Хохрайтер. «Быстрое и точное глубокое сетевое обучение с помощью экспоненциальных линейных единиц (ELU)» . В: CoRR abs/1511.07289 (2015). URL: <http://arxiv.org/abs/1511.07289> (цитируется на стр. 126).
- [105] Дональд Хебб. Организация поведения: нейропсихологическая теория. Уайли, 1949 год. (цитируется на стр. 128).
- [106] Кишан Мехротра, Чилукури К. Мохан и Санджай Ранка. Элементы искусственных нейронных сетей. Кембридж, Массачусетс, США: MIT Press, 1997. ISBN: 0-262-13328-8 (цитируется на страницах 128, 132).
- [107] Микель Олазаран. «Социологическое исследование официальной истории споров о перспектронах» . В: Социальные исследования науки 26.3 (1996), страницы 611–659. ISSN: 03063127. URL: <http://www.jstor.org/stable/285702> (цитируется на стр. 129).
- [108] Майкл Нильсен. Глава 2: Как работает алгоритм обратного распространения. <http://neuralnetworksanddeepl.com/chap2.html>. 2017 (цитируется на стр. 137, 141).
- [109] Мэтт Мазур. Пошаговый пример обратного распространения. <https://mattmazur.com/2015/17/03/a- шаг за шагом- обратное распространение-пример/>. 2015 (цитируется на стр. 137, 141).
- [110] Родриго Бененсон. Кто лучший в X? http://rodrigob.github.io/are_we_there_ еще/build/. 2017 (цитируется на стр. 163).
- [111] Джон Дучи, Элад Хазан и Йорам Сингер. «Адаптивные субградиентные методы для онлайн -обучения и стохастической оптимизации» . В: Дж. Max. Учиться. Рез. 12 (июль 2011 г.), страницы 2121–2159 . ISSN: 1532-4435 . URL-адрес: <http://dl.acm.org/citation.cfm?id=1953048&dl=2021068> (цитируется на стр. 164).
- [112] Мэттью Д. Зейлер. «ADADELTA: метод адаптивной скорости обучения» . В: CoRR abs/1212.5701 (2012). URL: <http://arxiv.org/abs/1212.5701> (цитируется на стр. 164).

- [113] Дидерик П. Кингма и Джимми Ба. «Адам: метод стохастической оптимизации» . В: CoRR abs/1412.6980 (2014). URL: <http://arxiv.org/abs/1412.6980> (цитируется на стр. 164, 311).
- [114] Грег Генрих. NVIDIA DIGITS: инициализация веса. <https://github.com/NVIDIA/DIGITS/blob/master/examples/weight-init/README.md>. 2015 (цитируется на стр. 165, 167).
- [115] Ксавье Глорот и Йошуа Бенжио. «Понимание сложности обучения нейронных сетей с глубокой прямой связью» . В: Материалы Международной конференции по искусственному интеллекту и статистике (AISTATS'10). Общество искусственного интеллекта и статистики. 2010 (цитируется на стр. 166).
- [116] Эндрю Джонс. Объяснение инициализации Xavier. <http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xavier-initialization>. 2016 (цитируется на стр. 166).
- [117] Кайминг Хе и др. «Углубление в выпрямители: превосходство на уровне человека в классификации ImageNet» . В: CoRR abs/1502.01852 (2015). URL: <http://arxiv.org/abs/1502.01852> (цитируется на стр. 167).
- [118] Авторы Keras. Инициализаторы Кераса. https://keras.io/initializers/#glorot_uniform. 2016 (цитируется на стр. 167).
- [119] Ричард Шелиски. Компьютерное зрение: алгоритмы и приложения. 1-й. Нью-Йорк, штат Нью-Йорк, США: Springer-Verlag New York, Inc., 2010. ISBN: 1848829345, 9781848829343 (цитируется на страницах 171, 177).
- [120] Виктор Пауэлл. Ядра изображений объясняются визуально. <http://setosa.io/ev/image-kernels/>. 2015 (цитируется на стр. 177).
- [121] Андрей Карпаты. Сверточные сети. <http://cs231n.github.io/convnets/> (цит. по стр. 186, 187, 191).
- [122] Йост Тобиас Спрингенберг и др. «Стремление к простоте: полностью сверточная сеть» . В: CoRR abs/1412.6806 (2014). URL: <http://arxiv.org/abs/1412.6806> (цитируется на стр. 188).
- [123] Сергей Иоффе и Кристиан Сегеди. «Пакетная нормализация: ускорение обучения глубокой сети за счет уменьшения внутреннего ковариантного сдвига» . В: CoRR abs/1502.03167 (2015). URL: <http://arxiv.org/abs/1502.03167> (цитируется на стр. 189, 193).
- [124] Франсуа Шолле. Б.Н. Вопросы (старые). <https://github.com/fchollet/keras/issues/1802#issuecomment-187966878> (цитируется на стр. 190).
- [125] Дмитрий Мишкин. CaffeNet-Benchmark – пакетная норма. <https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md> (цитируется на стр. 190).
- [126] Участники сообщества Reddit. Пакетная нормализация до или после ReLU? https://www.reddit.com/r/MachineLearning/comments/67gonq/d_batch_normalization_before_or_after_relu/ (цитируется на стр. 190).
- [127] Форрест Н. Яндоля и др. «SqueezeNet: точность уровня AlexNet с 50 раз меньшим количеством параметров и размером модели <1 МБ» . В: CoRR abs/1602.07360 (2016). URL: <http://arxiv.org/abs/1602.07360> (цитируется на стр. 192, 280, 281).
- [128] Пьер Сермане и др. «OverFeat: интегрированное распознавание, локализация и обнаружение с использованием сверточных сетей» . В: CoRR abs/1312.6229 (2013). URL: <http://arxiv.org/abs/1312.6229> (цитируется на стр. 229).

- [129] Андрей Карпаты. Нейронные сети (Часть I). <http://cs231n.github.io/neural-networks-1/> (цит. по стр. 253).
- [130] Кайминг Хе и соавт. «Отображение идентичности в глубоких остаточных сетях» . В: CoRR abs/1603.05027 (2016). URL: <http://arxiv.org/abs/1603.05027> (цитируется на стр. 279).
- [131] Кристиан Сегеди и др. «Переосмысление начальной архитектуры для компьютерного зрения» . В: CoRR abs/1512.00567 (2015). URL: <http://arxiv.org/abs/1512.00567> (цитируется на стр. 280).
- [132] Франсуа Шолле. «Xception: глубокое обучение с помощью глубоко отделимых сверток» . В: CoRR abs/1610.02357 (2016). URL: <http://arxiv.org/abs/1610.02357> (цитируется на стр. 280).
- [133] Википедия. E-ZPass. <https://en.wikipedia.org/wiki/E-ZPass> (цитируется на стр. 288).
- [134] Пол Виола и Майкл Джонс. «Быстрое обнаружение объектов с использованием расширенного каскада простых функций» . В: 2001, страницы 511–518 (цитируется на странице 313).