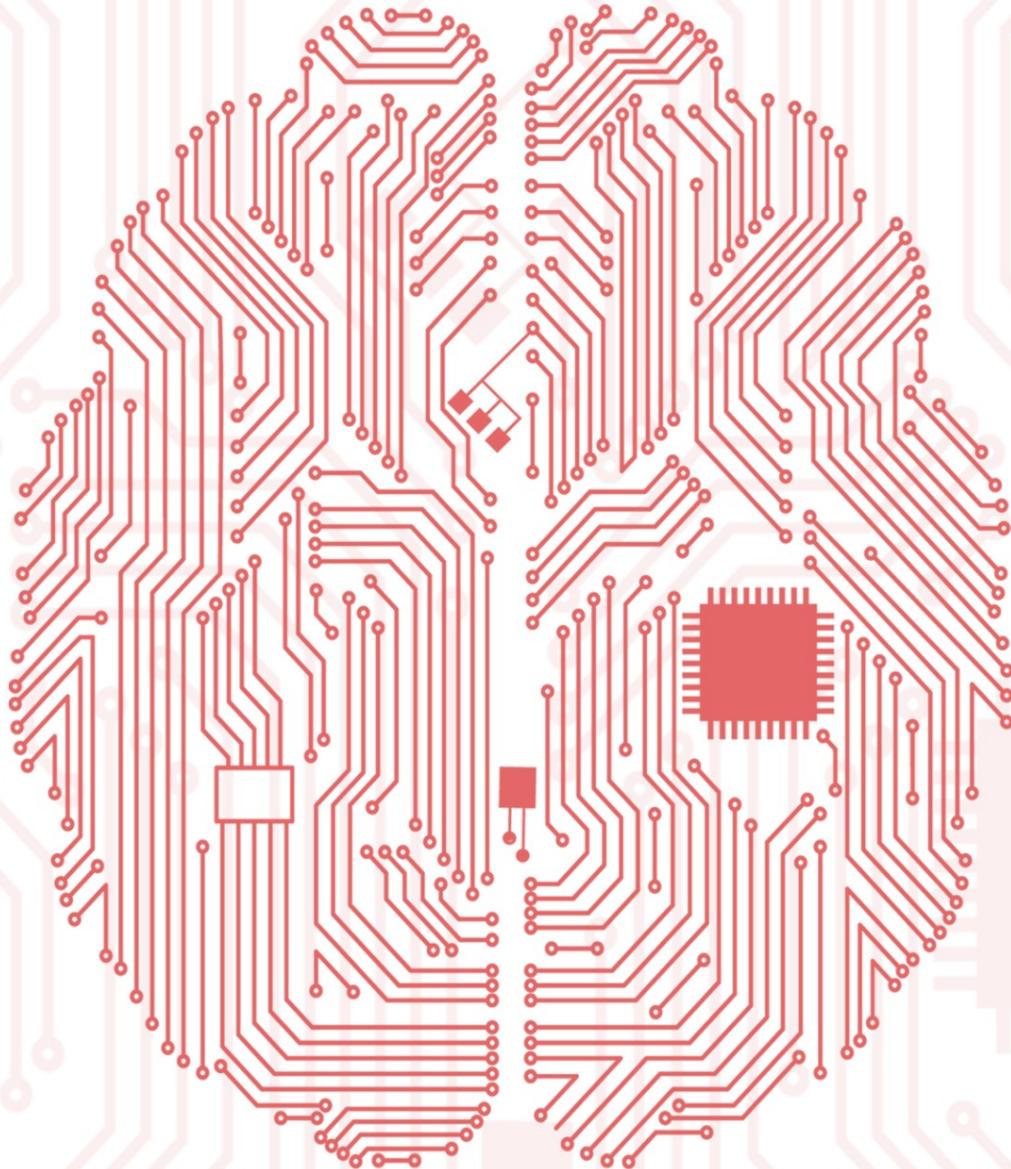


DEEP LEARNING FOR COMPUTER VISION



WITH PYTHON

Dr. Adrian Rosebrock

 PyImageSearch

Глубокое обучение для компьютерного зрения с ПИТОН

Набор «Практик»

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 странице <https://www.pyimagesearch.com/deep-learning-computer-vision .питон-книга/> 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 .	11
2	Увеличение данных .	13
2.1	Что такое увеличение данных?	13
2.2	Визуализация увеличения данных	14
2.3	Сравнение обучения с добавлением данных и без него 2.3.1 Набор данных Flowers-17 .	17
2.3.2	Предварительная обработка с учетом аспектов .	18
2.3.3	Цветы-17: без увеличения данных.	21
2.3.4	Цветы-17: с увеличением данных.	25
2.4	Резюме	29
3	Сети как экстракторы признаков .	31
3.1	Извлечение функций с помощью предварительно обученной CNN	32
3.1.1	Что такое HDF5? .	33
3.1.2	Запись функций в набор данных HDF5.	34
3.2	Процесс извлечения признаков 3.2.1	37
	Извлечение признаков из животных .	41
3.2.2	Извлечение признаков из CALTECH-101 .	42
3.2.3	Извлечение признаков из цветов-17. 3.3	42
	Обучение классификатора на извлеченных признаках	43
3.3.1	Результаты на животных.	45
3.3.2	Результаты CALTECH-101.	45
3.3.3	Результаты по Цветам-17.	46
3.4	Резюме	46

4	Понимание точности ранга 1 и ранга 5	49
4.1	Ранговая точность	49
4.1.1	Измерение точности 1-го и 5-го ранга.	51
4.1.2	Реализация ранжированной точности.	52
4.1.3	Ранжированная точность на цветах-17.	54
4.1.4	Ранжированная точность CALTECH-101.	54
4.2	Резюме	54
5	Тонкая настройка сетей	57
5.1	Перенос обучения и точная	57
5.1.1	Индексы и слои	60
5.1.2	Сетевая хирургия.	61
5.1.3	Тонкая настройка от начала до конца.	63
5.2	Резюме	69
6	Повышение точности с помощью сетевых ансамблей	71
6.1	Методы ансамбля	71
6.1.1	Неравенство Дженсена.	72
6.1.2	Построение ансамбля CNN.	73
6.1.3	Оценка ансамбля.	77
6.2	Резюме	80
7	продвинутых методов оптимизации	83
7.1	Методы адаптивной скорости	83
7.1.1	Adagrad	84
7.1.2	Ададельта.	84
7.1.3	RMSprop.	85
7.1.4	Адам.	85
7.1.5	Надам.	86
7.2	Выбор метода оптимизации 7.2.1 Три	86
	метода, которым вы должны научиться управлять: SGD, Adam и RMSprop. . 86	
7.3	Резюме	87
8	Оптимальный путь для применения глубокого обучения	89
8.1	Рецепт обучения 8.2	89
8.2	Перенос обучения или 8.3 Резюме	93
		94
9	Работа с HDF5 и большими наборами данных	95
9.1	Загрузка Kaggle: Dogs vs. Cats 9.2 Создание	95
	файла конфигурации 9.2.1 Ваш первый файл	96
	конфигурации .	97
9.3	Создание набора	98
	данных 9.4 Резюме	102

10 Соревнования в Kaggle: Dogs vs. Cats	103
10.1 Дополнительные препроцессоры изображений	103
10.1.1 Средняя предварительная обработка	104
10.1.2 Предварительная обработка патчей	105
10.1.3 Предварительная обработка урожая	107
10.2 Генераторы наборов данных HDF5	109
10.3 Внедрение AlexNet 10.4	112
Обучение AlexNet на Kaggle: Dogs vs. Cats 10.5 Оценка	117
AlexNet 10.6 Получение места в пятерке лучших в	120
таблице лидеров Kaggle 10.6.1 Извлечение функций с помощью	123
ResNet	123
10.6.2 Обучение классификатора логистической регрессии	127
10.7 Резюме	128
11 ГуглеНет	131
11.1 Начальный модуль (и его варианты)	132
11.1.1 Начало.	132
11.1.2 Миницепция.	133
11.2 MiniGoogLeNet на CIFAR-10 11.2.1	134
Реализация MiniGoogLeNet	135
11.2.2 Обучение и оценка MiniGoogLeNet на CIFAR-10.	140
11.2.3 MiniGoogLeNet: эксперимент №1	143
11.2.4 MiniGoogLeNet: Эксперимент №2	144
11.2.5 MiniGoogLeNet: эксперимент №3	145
11.3 Задача Tiny ImageNet 11.3.1 Загрузка	146
Tiny ImageNet	147
11.3.2 Крошечная структура каталогов ImageNet	147
11.3.3 Создание набора данных Tiny ImageNet.	148
11.4 DeeperGoogLeNet на Tiny ImageNet 11.4.1	153
Реализация DeeperGoogLeNet	153
11.4.2 Обучение DeeperGoogLeNet на Tiny ImageNet.	161
11.4.3 Создание сценария обучения.	161
11.4.4 Создание сценария оценки	163
11.4.5 Эксперименты DeeperGoogLeNet	165
11.5 Резюме	168
12 Ренет	171
12.1 ResNet и остаточный модуль 12.1.1	171
Углубление: остаточные модули и узкие места	172
12.1.2 Переосмысление остаточного модуля.	174
12.2 Внедрение ResNet	175
12.3 ResNet на CIFAR-10	180
12.3.1 Обучение ResNet на CIFAR-10 методом ctrl+c.	181
12.3.2 ResNet на CIFAR-10: Эксперимент №2.	185

12.4 Обучение ResNet на CIFAR-10 с уменьшением скорости обучения	188
12.5 ResNet на Tiny ImageNet 12.5.1 Обновление архитектуры ResNet .	192
12.5.2 Обучение ResNet на Tiny ImageNet методом ctrl+c.	193
12.5.3 Обучение ResNet на Tiny ImageNet с уменьшением скорости обучения.	194
12.6 Резюме	202

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 используйте эту ссылку: <http://pyimg.co/fnkxk> . Потратьте секунду, чтобы создать учетную запись сейчас, чтобы иметь доступ к дополнительным материалам.

пока вы работаете с книгой.

1. Введение

Добро пожаловать в Практический пакет глубокого обучения для компьютерного зрения с Python! Этот том должен стать следующим логическим шагом в вашем глубоком обучении компьютерному зрению после завершения Starter Bundle.

На этом этапе вы должны иметь четкое представление об основах параметризованного обучения, нейронных сетей и сверточных нейронных сетей (CNN). Вы также должны чувствовать себя относительно комфортно, используя библиотеку Keras и язык программирования Python для обучения собственных сетей глубокого обучения.

Цель набора «Практик» — развить ваши знания, полученные из набора «Начальный», и представить более продвинутые алгоритмы, концепции и профессиональные приемы — эти приемы будут рассмотрены в трех отдельных частях книги.

Первая часть будет посвящена методам, которые так или иначе используются для повышения точности вашей классификации. Один из способов повысить точность вашей классификации — применить методы трансферного обучения, такие как тонкая настройка или использование вашей сети в качестве средства извлечения признаков.

Мы также рассмотрим ансамблевые методы (т. е. обучение нескольких сетей и объединение результатов) и то, как эти методы могут дать вам хороший прирост классификации с небольшими дополнительными усилиями. Методы регуляризации, такие как увеличение данных, используются для создания дополнительных обучающих данных — почти во всех ситуациях увеличение данных улучшает способность вашей модели к обобщению. Более продвинутые алгоритмы оптимизации, такие как Adam [1], RMSprop [2] и другие, также могут использоваться для некоторых наборов данных, чтобы снизить потери. После того, как мы рассмотрим эти методы, мы рассмотрим оптимальный путь их применения, чтобы гарантировать, что вы получите максимальную пользу с наименьшими усилиями.

Затем мы переходим ко второй части пакета Practitioner Bundle, в которой основное внимание уделяется большим наборам данных и более экзотическим сетевым архитектурам. До сих пор мы работали только с наборами данных, которые поместились в основную память нашей системы, но что, если наш набор данных слишком велик, чтобы поместиться в ОЗУ? Что мы делаем тогда? Мы обратимся к этому вопросу в главе 9, когда будем работать с HDF5.

Учитывая, что мы будем работать с большими наборами данных, мы также сможем обсудить более сложные сетевые архитектуры с использованием AlexNet, GoogLeNet, ResNet и более глубоких вариантов VGGNet. Эти сетевые архитектуры будут применяться к более сложным наборам данных и соревнованиям, включая

Kaggle: испытание на распознавание собак против кошек [3], а также испытание cs231n Tiny ImageNet [4], точно такое же задание, в котором соревнуются студенты Stanford CNN. 25-е место в таблице лидеров Kaggle Dogs vs. Cats и первое место в соревновании cs231n для нашего типа техники.

В заключительной части этой книги рассматриваются приложения глубокого обучения для компьютерного зрения за пределами классификации изображений, включая базовое обнаружение объектов, глубокое сновидение и нейронный стиль, генеративно-состязательные сети (GAN) и суперразрешение изображений. Опять же, методы, описанные в этом томе, должны быть намного более продвинутыми, чем в Starter Bundle — здесь вы начнете отделять себя от новичка в области глубокого обучения и превратитесь в настоящего практикующего специалиста по глубокому обучению. Чтобы начать свое превращение в эксперта по глубокому обучению, просто переверните страницу.

2. Увеличение данных

Согласно Гудфелло и др., регуляризация — это «любая модификация, которую мы вносим в алгоритм обучения, направленная на уменьшение ошибки обобщения, но не ошибки обучения» [5]. Короче говоря, регуляризация стремится уменьшить нашу ошибку тестирования, возможно, за счет небольшого увеличения ошибки обучения.

Мы уже рассматривали различные формы регуляризации в главе 9 Starter Bundle; однако это были параметризованные формы регуляризации, требующие от нас обновления нашей функции потери/обновления. На самом деле существуют и другие типы регуляризации, которые: 1. Изменяют саму сетевую архитектуру.

2. Дополнить данные, переданные в сеть для обучения.

Dropout — отличный пример модификации сетевой архитектуры за счет достижения большей универсальности. Здесь мы вставляем слой, который случайным образом отключает узлы от предыдущего слоя к следующему слою, тем самым гарантируя, что ни один узел не отвечает за изучение того, как представлять данный класс.

В оставшейся части этой главы мы обсудим другой тип регуляризации, называемый увеличением данных. Этот метод намеренно искажает обучающие примеры, слегка меняя их внешний вид, прежде чем передать их в сеть для обучения. Конечным результатом является то, что сеть постоянно видит «новые» точки обучающих данных, сгенерированные из исходных обучающих данных, что частично избавляет нас от необходимости собирать больше обучающих данных (хотя в целом сбор дополнительных обучающих данных редко вредит вашему алгоритму).

2.1 Что такое увеличение данных?

Расширение данных включает в себя широкий спектр методов, используемых для создания новых обучающих выборок из исходных путем применения случайных флуктуаций и возмущений таким образом, чтобы метки классов не менялись. Наша цель при применении аугментации данных — повысить обобщаемость модели. Учитывая, что наша сеть постоянно видит новые, слегка измененные версии точек входных данных, она способна изучать более надежные функции. Во время тестирования мы не применяем увеличение данных и оцениваем нашу обученную сеть — в большинстве случаев вы увидите повышение точности тестирования, возможно, за счет небольшого снижения точности обучения.

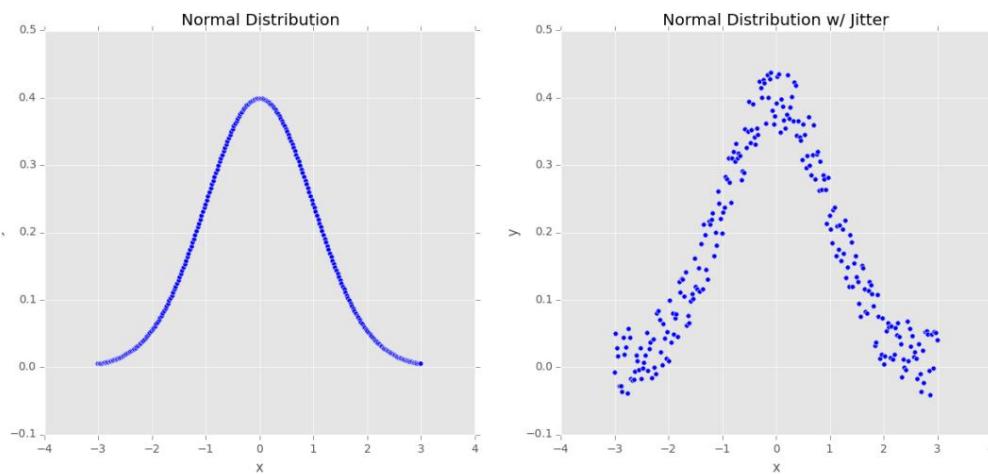


Рисунок 2.1: Слева: выборка из 250 точек данных, которые точно соответствуют нормальному распределению. Справа: добавление небольшого количества случайного «джиттера» в распределение. Этот тип увеличения данных может повысить обобщаемость наших сетей.

Давайте рассмотрим рисунок 2.1 (слева) нормального распределения с нулевым средним и единичной дисперсией. Обучение модели машинного обучения на этих данных может привести к точному моделированию распределения, однако в реальных приложениях данные редко следуют такому точному распределению.

Вместо этого, чтобы повысить обобщаемость нашего классификатора, мы можем сначала случайным образом изменить точки распределения, добавив некоторые значения ϵ , взятые из случайного распределения (справа). Наш график по-прежнему следует приблизительно нормальному распределению, но это не идеальное распределение, как показано слева. Модель, обученная на этих данных, с большей вероятностью будет обобщать примеры точек данных, не включенных в обучающий набор.

В контексте компьютерного зрения расширение данных представляется естественным. Например, мы можем получить дополнительные обучающие данные из исходных изображений, применив простые геометрические преобразования, такие как случайные:

1. Переводы
2. Повороты 3.

Изменения масштаба

4. Сдвиг 5.

Горизонтальные (и в некоторых случаях вертикальные)

перевороты Применение (небольшого) количества этих преобразований к входному изображению слегка изменит его внешний вид, но не изменит метку класса — тем самым делая увеличение данных очень естественным и простым методом для применения к глубокому обучению для задач компьютерного зрения. Более продвинутые методы увеличения данных, применяемые к компьютерному зрению, включают случайное изменение цветов в заданном цветовом пространстве [6] и нелинейные геометрические искажения [7].

2.2 Визуализация увеличения данных

Лучший способ понять увеличение данных применительно к компьютерным задачам — это просто визуализировать увеличение и искажение заданного ввода. Чтобы выполнить эту визуализацию, давайте создадим простой скрипт Python, который использует встроенную мощь Keras для выполнения увеличения данных. Создайте новый файл, назовите его `augmentation_demo.py`. и вставьте следующий код:

```

1 # импортируем необходимые пакеты 2
из keras.preprocessing.image import ImageDataGenerator 3 из
keras.preprocessing.image import img_to_array 4 из keras.preprocessing.image
import load_img 5 import numpy as np 6 import argparse

```

Строки 2-6 импортируют необходимые пакеты Python. Обратите внимание на строку 2 , где мы импортируем класс ImageDataGenerator из Keras — этот код будет использоваться для увеличения данных и включает все соответствующие методы, которые помогут нам преобразовать наше входное изображение.

Далее мы анализируем наши аргументы командной строки:

```

8 # построить разбор аргумента и разобрать аргументы 9 ap =
argparse.ArgumentParser() 10 ap.add_argument("-i", "--image", required=True,
11     help="путь к входному изображению")
12 ap.add_argument("-o", "--output", required=True,
13     help="путь к выходному каталогу для хранения примеров дополнений")
14 ap.add_argument("-p", "--prefix", type=str, default="image", help="префикс
15     выходного имени файла") 16 args = vars(ap.parse_args())

```

Наш сценарий требует трех аргументов командной строки, каждый из которых подробно

описан ниже: • --image: это путь к входному изображению, к которому мы хотим применить дополнение данных и визуализировать результаты.

- --output: после применения увеличения данных к заданному изображению мы хотели бы сохранить результат на диск, чтобы мы могли его проверить — этот переключатель управляет выходным каталогом.

- --prefix: строка, которая будет добавлена к имени выходного файла изображения.

Теперь, когда наши аргументы командной строки проанализированы, давайте загрузим наше входное изображение, преобразуем его в массив, совместимый с Keras, и добавим к изображению дополнительное измерение, как если бы мы готовили наше изображение для классификации:

```

18 # загрузите входное изображение , преобразуйте его в массив NumPy, а затем
19 # измените его форму, чтобы оно имело дополнительное измерение ]) 22
изображение = img_to_array (изображение) 23 изображение = np.expand_dims
(изображение, ось = 0)

```

Теперь мы готовы инициализировать наш ImageDataGenerator:

```

25 # создать генератор изображений для увеличения данных, затем 26 #
инициализировать общее количество сгенерированных изображений 27 aug
= ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
28     height_shift_range = 0.1, shear_range = 0.2, zoom_range = 0.2,
29     horizontal_flip = True, fill_mode = «ближайший»)
всего 30 = 0

```

Класс ImageDataGenerator имеет ряд параметров, слишком много для перечисления в этой книге. Полный обзор параметров см. в официальной документации Keras (<http://pyimg.co/j8ad8>).

Вместо этого мы сосредоточимся на параметрах дополнения, которые вы, скорее всего, будете использовать в своих собственных приложениях. Параметр `rotate_range` управляет диапазоном градусов случайного вращения. Здесь мы позволим нашему входному изображению произвольно поворачиваться на ± 30 градусов. И `width_shift_range`, и `height_shift_range` используются для горизонтального и вертикального сдвига соответственно. Значение параметра представляет собой долю заданного измерения, в данном случае 10%.

Параметр `shear_range` управляет углом в радианах против часовой стрелки, на который можно срезать наше изображение. Затем у нас есть `zoom_range`, значение с плавающей запятой, которое позволяет «увеличивать» или «уменьшать масштаб» изображения в соответствии со следующим равномерным распределением значений: [1 - `zoom_range`, 1 + `zoom_range`].

Наконец, логическое значение `horizontal_flip` определяет, разрешено ли переворачивать входные данные по горизонтали в процессе обучения. Для большинства приложений компьютерного зрения горизонтальное отражение изображения не изменяет результирующую метку класса, но есть приложения, в которых горизонтальное (или вертикальное) отражение изменяет семантическое значение изображения. Будьте осторожны при применении этого типа увеличения данных, поскольку наша цель — немного изменить входное изображение, тем самым сгенерировав новую обучающую выборку, не изменяя саму метку класса. Для более подробного обзора преобразований изображений обратитесь к модулю № 1 в PyImageSearch Gurus ([8], [PyImageSearch Гурзу](#)) а также Шелиски [9].

Как только `ImageDataGenerator` инициализирован, мы можем создать новые обучающие примеры:

```

32 # создать фактический генератор Python 33
print("[INFO] генерирует изображения...") 34 imageGen =
35     aug.flow(image, batch_size=1, save_to_dir=args["output"], save_prefix=args["prefix"],
36             save_format="jpg")
37 # перебираем примеры из нашего генератора увеличения данных изображения 38
38     # для изображения в imageGen: # увеличиваем общее количество счетчиков += 1
39
40
41
42     # если мы достигли 10 примеров, выйти из цикла , если всего == 10:
43
44     переменна

```

Строки 34 и 35 инициализируют генератор Python, используемый для создания наших дополненных изображений. Мы передадим наше входное изображение, размер пакета , равный 1 (поскольку мы увеличиваем только одно изображение), вместе с несколькими дополнительными параметрами, чтобы указать пути к файлам выходного изображения, префикс для каждого пути к файлу и формат файла изображения. Затем строка 38 начинает перебирать каждое изображение в генераторе `imageGen`. Внутри `imageGen` автоматически генерирует новую обучающую выборку каждый раз, когда она запрашивается через цикл. Затем мы увеличиваем общее количество примеров расширения данных, записанных на диск, и останавливаем выполнение скрипта, как только мы достигаем десяти примеров.

Чтобы визуализировать увеличение данных в действии, мы будем использовать рисунок 2.2 (слева), изображение Джеммы, моей семейной гончей. Чтобы сгенерировать новые тренировочные образы Джеммы, просто выполните следующую команду:

```
$ python augmentation_demo.py --image jemma.png --выходной вывод
```

После выполнения скрипта вы должны увидеть десять изображений в выходном каталоге:

```
$ ls output/
image_0_1227.jpg image_0_2358.jpg image_0_4205.jpg image_0_4770.jpg
```

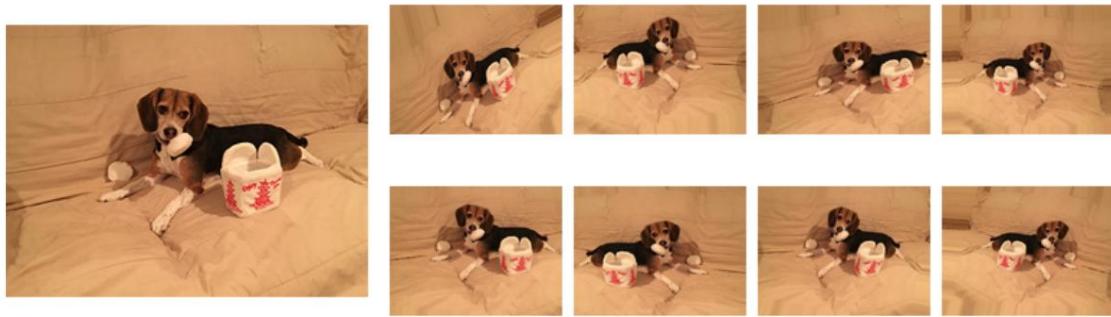


Рисунок 2.2: Слева: входное изображение, к которому мы собираемся применить увеличение данных. Справа: монтаж примеров увеличения данных. Обратите внимание, как каждое изображение было случайным образом повернуто, обрезано, увеличено и отражено по горизонтали.

[image_0_1933.jpg](#) [image_0_2914.jpg](#) [image_0_4657.jpg](#) [image_0_6934.jpg](#) [image_0_9197.jpg](#)
[image_0_953.jpg](#)

Я построил монтаж каждого из этих изображений, чтобы вы могли представить их на рис. 2.2 (справа). Обратите внимание, как каждое изображение было случайным образом повернуто, обрезано, увеличено и отражено по горизонтали . В каждом случае изображение сохраняет исходную метку класса: собака; однако каждое изображение было немного изменено, что дало нашей нейронной сети новые шаблоны для обучения при обучении. Поскольку входные изображения будут постоянно меняться (в то время как метки классов остаются прежними), точность нашего обучения часто снижается по сравнению с обучением без увеличения данных.

Однако, как мы узнаем позже в этой главе, аугментация данных может помочь значительно уменьшить переоснащение, в то же время гарантируя, что наша модель лучше обобщает новые входные выборки. Более того , при работе с наборами данных, где у нас слишком мало примеров для применения глубокого обучения, мы можем использовать увеличение данных для создания дополнительных обучающих данных, тем самым уменьшая количество размеченных вручную данных, необходимых для обучения сети глубокого обучения.

2.3 Сравнение обучения с дополнением данных и без него

В первой части этого раздела мы обсудим набор данных Flowers-17, очень маленький набор данных (с точки зрения глубокого обучения для задач компьютерного зрения), и то, как увеличение данных может помочь нам искусственно увеличить размер этого набора данных путем создания дополнительные обучающие образцы. Далее мы проведем два эксперимента: 1. Обучим MiniVGGNet на Цветах-17 без увеличения данных.

2. Обучить MiniVGGNet на Цветах-17 с аугментацией данных.

Как мы увидим, применение аугментации данных значительно уменьшает переоснащение и позволяет MiniVGGNet получать значительно более высокую точность классификации.

2.3.1 Набор данных Flowers-17

Набор данных Flowers-17 [10] представляет собой задачу мелкозернистой классификации, в которой наша задача состоит в том, чтобы распознать 17 различных видов цветов. Набор данных изображений довольно мал, всего 80 изображений на класс , всего 1360 изображений. Общее эмпирическое правило при применении глубокого обучения к задачам компьютерного зрения состоит в том, чтобы иметь 1000-5000 примеров на класс, поэтому здесь у нас, безусловно, огромный дефицит.

Мы называем «Цветы-17» задачей мелкозернистой классификации, потому что все категории очень похожи (т. е. виды цветов). На самом деле, мы можем думать о каждой из этих категорий как о подкатегориях. Категории , безусловно, разные, но имеют много общего в структуре (например, лепестки,



Рисунок 2.3: Выборка из пяти (из семнадцати) классов в наборе данных Flowers-17, где каждый класс представляет определенный вид цветка.

тычинка, пестик и др.). Задачи мелкоузернистой классификации, как правило, являются наиболее сложными для глубокого анализа. обучающихся практиков, так как это означает, что наши модели машинного обучения должны изучать чрезвычайно отличительные черты, чтобы различать очень похожие классы. Эта узкая классификация задача становится еще более проблематичной, учитывая наши ограниченные обучающие данные.

2.3.2 Предварительная обработка с учетом аспектов

До этого момента мы только предварительно обрабатывали изображения, изменяя их размер до фиксированного размера, игнорируя соотношение сторон. В некоторых ситуациях, особенно для базовых наборов эталонных данных, это допустимо.

Однако для более сложных наборов данных мы все равно должны стремиться изменить размер до фиксированного размера, но сохранить соотношение сторон. Чтобы визуализировать это действие, рассмотрите рисунок 2.4.

Слева у нас есть входное изображение, которое нам нужно изменить до фиксированной ширины и высоты. Игнорирование соотношения сторон, мы изменяем размер изображения до 256×256 пикселей (посередине), эффективно сжимая и искажение изображения таким образом, чтобы оно соответствовало нашим желаемым размерам. Лучшим подходом было бы примите во внимание соотношение сторон изображения (справа), где мы сначала изменяем размер по более короткому размер так, чтобы ширина была 256 пикселей, а затем обрезать изображение по высоте, чтобы высота 256 пикселей.

Хотя мы фактически отбросили часть изображения во время обрезки, мы также сохранили исходное соотношение сторон изображения. Поддержание постоянного соотношения сторон позволяет нашей сверточной Нейронной сети, чтобы узнать больше отличительных, последовательных функций. Это распространенная техника, мы будем применять при работе с более продвинутыми наборами данных в остальной части Практика.

Пакет и пакет ImageNet.

Чтобы увидеть, как реализована предварительная обработка с учетом аспектов, давайте обновим наш pyimagesearch. структура проекта для включения AspectAwarePreprocessor:

```
--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- nn
| |--- предварительная обработка
| | |--- __init__.py
| | |--- аспектаварепропроцессор.py
```

2.3 Сравнение обучения с дополнением данных и без него

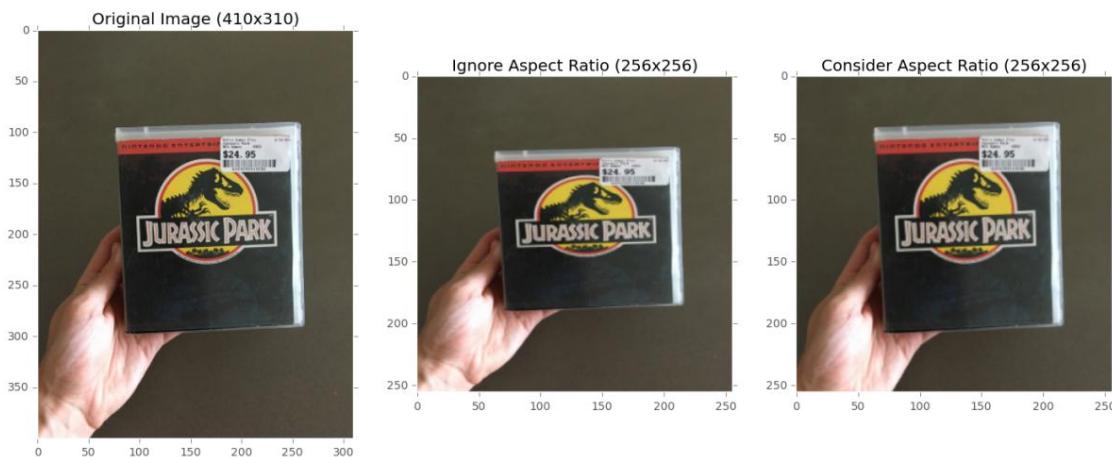


Рисунок 2.4: Слева: Исходное входное изображение (410× 310). Посередине: изменение размера изображения до 256× 256 . пикселей без учета соотношения сторон. Обратите внимание, как теперь изображение кажется сплющенным и искаженным. Верно: Изменение размера изображения до 256× 256 с сохранением соотношения сторон.

```
| |--- imagettoArraypreprocessor.py
| |--- simplepreprocessor.py
| --- утилиты
```

Обратите внимание, как мы добавили новый файл с именем aspectawarepreprocessor.py внутри подмодуль предварительной обработки — в этом месте будет работать наш новый препроцессор. Открыть aspectawarepreprocessor.py и вставьте следующий код:

```
1 # импортируем необходимые пакеты
2 импорта imutils
3 импорт cv2
4
5 класс AspectAwarePreprocessor:
6     def __init__(я, ширина, высота, интер=cv2.INTER_AREA):
7         # сохранить ширину, высоту и интерполяцию целевого изображения
8         # метод, используемый при изменении размера
9         собственная ширина = ширина
10        self.height = высота
11        self.inter = интер
```

Так же, как и в нашем SimplePreprocessor, наш конструктор требует два параметра (желаемый ширина и высота целевого выходного изображения) вместе с методом интерполяции, используемым при изменение размера изображения. Затем мы можем определить функцию предварительной обработки ниже:

```
13     предварительная обработка def (я, изображение):
14         # получить размеры изображения, а затем инициализировать
15         # дельты для использования при кадрировании
16         (ч, ш) = изображение.форма[:2]
17         dB = 0
18         дН = 0
```

Функция предварительной обработки принимает единственный аргумент — изображение , которое мы хотим предварительно обработать. Строки 16 захватывает ширину и высоту входного изображения, а строки 17 и 18 определяют дельта-смещения, которые мы будем использовать при кадрировании по большему измерению. Опять же, наш аспект-осведомленный препроцессор представляет собой двухэтапный алгоритм:

1. Шаг №1: Определите кратчайшее измерение и измените размер вдоль него.
2. Шаг № 2: Обрежьте изображение по наибольшему измерению, чтобы получить целевые ширину и высоту.

Следующий блок кода проверяет, меньше ли ширина высоты, и если да, то изменяет размеры по ширине:

```

20      # если ширина меньше высоты, то изменить размер
21      # по ширине (т.е. по меньшему измерению) и затем
22      # обновить дельты, чтобы обрезать высоту до нужной
23      # измерение
24      если ш < ч:
25          изображение = imutils.resize (изображение, ширина = self.width,
26                                      интер=себя.интер)
27          dH = int((image.shape[0] - self.height) / 2.0)

```

В противном случае, если высота меньше ширины, то ресайзим по высоте:

```

29      # в противном случае высота меньше ширины, поэтому
30      # изменить размер по высоте, а затем обновить дельты
31      # обрезать по ширине
32      еще:
33          изображение = imutils.resize (изображение, высота = self.height,
34                                      интер=себя.интер)
35          dW = int((image.shape[1] - собственная ширина) / 2.0)

```

Теперь, когда наше изображение изменено, нам нужно повторно захватить ширину и высоту и использовать дельты для обрезать центр изображения:

```

37      # теперь, когда размеры наших изображений изменены, нам нужно
38      # повторно захватить ширину и высоту, а затем выполнить
39      # урожай
40      (ч, ш) = изображение.форма[:2]
41      изображение = изображение[dH:h - dH, dW:w - dW]
42
43      # наконец, измените размер изображения на предоставленный пространственный
44      # размеры, чтобы наше выходное изображение всегда было фиксированным
45      # размер
46      вернуть cv2.resize (изображение, (self.width, self.height),
47                          интерполяция = self.inter)

```

При обрезке (из-за ошибок округления) размеры целевого изображения нашего изображения могут отличаться на ± один пиксель; поэтому мы вызываем cv2.resize , чтобы убедиться, что наше выходное изображение имеет желаемый вид. ширина и высота. Затем предварительно обработанное изображение возвращается вызывающей функции. Теперь, когда мы реализовали наш препроцессор AspectAwarePreprocessor, давайте задействуем его при обучении MiniVGGNet архитектура на наборе данных Flowers-17.

2.3.3 Цветы-17: без увеличения данных

Для начала давайте установим базовый уровень без увеличения данных при обучении архитектуры MiniVGGNet (глава 15, Starter Bundle) на наборе данных Flowers-17. Откройте новый файл, назовите его minivggnet_flowers17.py, и мы приступим к работе:

```
1 # импортируем необходимые пакеты 2
из sklearn.preprocessing импортируем LabelBinarizer 3 из
sklearn.model_selection import train_test_split 4 из sklearn.metrics _ _ _
--- pyimagesearch.nn.conv импортировать MiniVGGNet
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 14 импортировать os
```

Строки 2-14 импортируют необходимые пакеты Python. Большинство этих импортов вы видели раньше, но я хочу обратить ваше внимание на:

1. Стока 6: здесь мы импортируем только что определенный AspectAwarePreprocessor.
2. Стока 7: несмотря на использование отдельного препроцессора изображений, мы по-прежнему сможем использовать SimpleDatasetLoader для загрузки нашего набора данных с диска.
3. Стока 8: мы будем обучать архитектуру MiniVGGNet на нашем наборе данных.

Далее мы анализируем 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 args = vars(ap.parse_args())
```

Здесь нам нужен только один переключатель, --dataset, который является путем к нашему каталогу набора данных Flowers-17 на диске.

Давайте продолжим и извлечем метки классов из наших входных изображений:

```
22 # получить список изображений, которые мы будем описывать, затем извлечь 23
# имена меток классов из путей к изображениям 24 print("[INFO] loading images...")
25 imagePaths = list(paths.list_images(args ["набор данных"])) 26 classNames =
[pt.split(os.path.sep)[-2] для pt в imagePaths] 27 classNames = [str(x) для x в
np.unique(classNames)]
```

Наш набор данных Flowers-17 имеет следующую структуру каталогов:

цветы17/{виды}/{изображение}

Ниже приведен пример изображения в наборе данных:

flowers17/колокольчик/image_0241.jpg

Следовательно, чтобы извлечь метки классов, мы можем просто извлечь предпоследний индекс после разделения по разделителю путей (строка 26), получив текстовый колокольчик. Если вам трудно понять, как работает это извлечение путей и меток, я бы посоветовал открыть оболочку Python и поиграть с путями к файлам и разделителями путей. В частности, обратите внимание, как вы можете разделить строку на основе разделителя пути операционной системы, а затем использовать индексацию Python для извлечения различных частей массива. Затем в строке 27 определяется уникальный набор меток классов (в данном случае всего 17 классов) из путей к изображениям.

Учитывая наши imagePaths, мы можем загрузить набор данных Flowers-17 с диска:

```
29 # инициализируем препроцессоры
изображений 30 aap = AspectAwarePreprocessor(64,
64) 31 iap = ImageToArrayPreprocessor()
32
33 # загрузить набор данных с диска, затем масштабировать необработанные
интенсивности пикселей 34 # до диапазона [0, 1] 35 sdl =
SimpleDatasetLoader(preprocessors=[aap, iap]) 36 (data, labels) = sdl.load(imagePaths,
verbose =500) 37 данных = data.astype("с плавающей запятой") / 255.0
```

Строка 30 инициализирует наш препроцессор AspectAwarePreprocessor таким образом, что каждое обрабатываемое им изображение будет иметь размер 64× 64 пикселя. Затем ImageToArrayPreprocessor инициализируется в строке 31, что позволяет нам преобразовывать изображения в массивы, совместимые с Keras. Затем мы создаем экземпляр SimpleDatasetLoader , используя эти два препроцессора соответственно (строка 35).

Данные и соответствующие метки загружаются с диска в строке 36. Все изображения в данных затем массив нормализуется до диапазона [0,1] путем деления необработанных интенсивностей пикселей на 255.

Теперь, когда наши данные загружены, мы можем выполнить разделение обучения и тестирования (75 процентов для обучения, 25 процентов на тестирование) вместе с одноразовым кодированием наших этикеток:

```
39 # разбить данные на обучающие и тестовые сплиты, используя 75% от 40 #
данные для обучения и оставшиеся 25% для тестирования 41 (trainX, testX, trainY,
testY) = train_test_split(data, labels, test_size=0.25, random_state= 42)
42
43
44 # преобразовать метки из целых чисел в векторы 45 trainY = LabelBinarizer().fit_transform(trainY)
46 testY = LabelBinarizer().fit_transform(testY)
```

Для обучения нашего классификатора цветов мы будем использовать архитектуру MiniVGGNet вместе с оптимизатором SGD:

```
48 # инициализируем оптимизатор и модель
49 print("[INFO] компилируем модель...") 50 opt
= SGD(lr=0.05) 51 model = MiniVGGNet.build(width=64,
height=64, depth=3, классы =len(classNames)) 53
52     model.compile(loss="categorical_crossentropy",
оптимизатор=opt,
54     метрики=["точность"])
```

```

55
56 # обучаем сеть
57 print("[INFO] обучающая сеть...")
58 H = model.fit(trainX, trainY, validation_data=(testX, testY),
59     batch_size=32, эпохи=100, подробный=1)

```

Архитектура MiniVGGNet будет принимать изображения с пространственными размерами $64 \times 64 \times 3$ (64 пикселей в ширину, 64 пикселя в высоту и 3 канала). Общее количество классов равно `len(classNames)` что в данном случае равно семнадцати, по одному для каждой категории в наборе данных «Цветы-17».

Мы будем обучать MiniVGGNet с использованием SGD с начальной скоростью обучения $\alpha = 0,05$. Мы намеренно не учитываем снижение скорости обучения, чтобы мы могли продемонстрировать влияние увеличения данных в следующем разделе. Линии 58 и 59 обучают MiniVGGNet в общей сложности 100 эпох.

Затем мы оцениваем нашу сеть и строим графики потерь и точности с течением времени:

```

61 # оценить сеть
62 print("[INFO] оценка сети...")
63 прогноза = model.predict(testX, batch_size=32)
64 печать (classification_report (testY.argmax (ось = 1),
65     прогнозы.argmax (ось = 1), target_names = classNames))
66
67 # график потери и точности обучения
68 plt.style.use("ggplot")
69 пл.цифра()
70 plt.plot(np.arange(0, 100), H.history["потеря"], метка="train_loss")
71 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
72 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
73 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
74 plt.title ("«Тренировочные потери и точность»")
75 plt.xlabel("Эпоха #")
76 plt.ylabel("Потери/точность")
77 табл.легенда()
78 пл.показать()

```

Чтобы получить базовую точность на Flowers-17 с помощью MiniVGGNet, просто выполните следующее команда:

```

$ python minivggnet_flowers17.py --dataset ../flowers17/images
[INFO] загрузка изображений...
[INFO] обработано 500/1360
[INFO] обработано 1000/1360
[INFO] компиляция модели...
[INFO] тренировочная сеть...
Обучение на 1020 образцах, проверка на 340 образцах
Эпоха 1/100
...
Эпоха 100/100
3s - убыток: 0,0030 - акк: 1,0000 - val_loss: 1,7683 - val_acc: 0,6206
[INFO] оценка сети...
      точность      вспомнить поддержку f1-score
    колокольчик      0,48      0,67      0,56      18
    лютик мать-        0,67      0,60      0,63      20
    и-мачеха        0,53      0,40      0,46      20

```

первоцвет	0,35	0,44	0,39	18
крокус	0,62	0,50	0,56	20
нарцисс	0,43	0,33	0,38	27
	0,74	0,85	0,79	20
ромашка одуванчик	0,61	0,83	0,70	23
рябчатый ирис	0,72	0,82	0,77	22
	0,80	0,76	0,78	21
lilyvalley	0,59	0,67	0,62	15
	0,82	0,74	0,78	19
анютины	0,64	0,39	0,49	23
глазки	0,95	0,91	0,93	23
подснежник	0,88	0,74	0,80	19
	0,18	0,25	0,21	16
подсолнух тигровая лилия	0,60	0,62	0,65	16
подсолнух ветреное	0,62	0,62	0,62	16
среднее / общее	0,64	0,62	0,62	340

Как видно из вывода, мы можем получить 64-процентную точность классификации, что составляет довольно разумно, учитывая наш ограниченный объем обучающих данных. Тем не менее, что касается нашей потери и график точности (рисунок 2.5). Как видно из графика, наша сеть быстро начинает эпоха 20. Причина такого поведения в том, что у нас есть только 1020 обучающих примеров с 60 изображений для каждого класса (остальные изображения используются для тестирования). Имейте в виду, что в идеале мы должны иметь где-то между 1000-5000 примеров на класс при обучении сверточной нейронной сети.

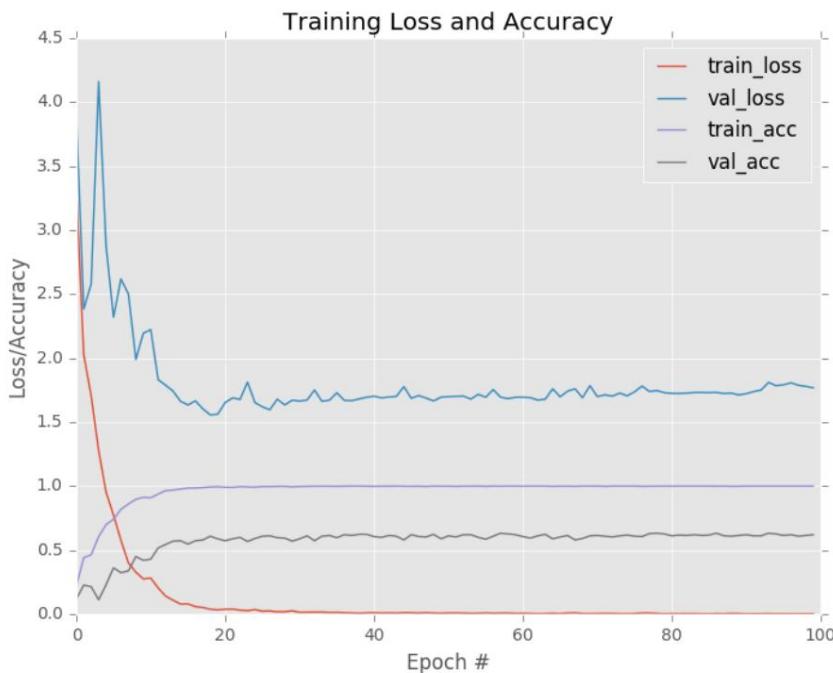


Рисунок 2.5: График обучения для MiniVGGNet, примененный к набору данных Flowers-17 без увеличения данных . Обратите внимание, как переоснащение начинает происходить после эпохи 25 по мере увеличения наших потерь при проверке.

Кроме того, точность обучения взлетает до 95% в первые несколько эпох, в конечном итоге достигая 100% точность в более поздние эпохи — этот результат является явным случаем переобучения. Из-за отсутствия существенные обучающие данные, MiniVGGNet моделирует основные закономерности в обучающих данных

слишком близко и не может обобщить тестовые данные. Чтобы бороться с переоснащением, мы можем применить методы регуляризации — в контексте этой главы нашим методом регуляризации будет увеличение данных. На практике вы также должны включать другие формы регуляризации (уменьшение веса, отсев и т. д.), чтобы еще больше уменьшить влияние переобучения.

2.3.4 Цветы-17: с увеличением данных

В этом примере мы собираемся применить пример того же процесса обучения, что и в предыдущем разделе, только с одним дополнением: мы будем применять увеличение данных. Чтобы увидеть, как увеличение данных может повысить точность нашей классификации при предотвращении переобучения, откройте новый файл, назовите его `minivggnet_flowers17_data_aug.py` и приступайте к работе:

```
1 # импортируем необходимые пакеты 2
из sklearn.preprocessing импортируем LabelBinarizer 3 из
sklearn.model_selection import train_test_split 4 из sklearn.metrics __
----- pyimagesearch.nn.conv импортировать MiniVGGNet
9 из keras.preprocessing.image импортировать ImageDataGenerator 10 из
keras.optimizers импортировать SGD 11 из imutils импортировать пути 12
импортировать matplotlib.pyplot как plt 13 импортировать numpy как np 14
импортировать argparse 15 импортировать ОС
```

Наш импорт такой же, как и в `minivggnet_flowers17.py`, за исключением `Line`

9 , где мы импортируем класс `ImageDataGenerator`, используемый для увеличения данных.

Затем давайте проанализируем наши аргументы командной строки и извлечем имена классов из путей к изображениям:

```
17 # построить разбор аргумента и разобрать аргументы 18 ap =  
argparse.ArgumentParser() 19 ap.add_argument("-d", "--dataset",  
required=True,  
20         help="путь к входному набору  
данных") 21 args = vars(ap.parse_args())  
22  
23 # получить список изображений, которые мы будем описывать, затем извлечь 24  
# имена меток классов из путей к изображениям 25 print("[INFO] loading images...")  
26 imagePaths = list(paths.list_images(args["набор данных"])) 27 classNames =  
[pt.split(os.path.sep)[-2] для pt в imagePaths] 28 classNames = [str(x) для x в  
np.unique(classNames)]
```

А также загрузим наш набор данных с диска, создадим наши сплиты для обучения/тестирования и закодируем наши метки:

30 # инициализируем препроцессоры
изображений 31 aap = AspectAwarePreprocessor(64,
64) 32 iap = ImageToArrayPreprocessor()

```

34 # загрузить набор данных с диска, затем масштабировать необработанные
интенсивности пикселей 35 # до диапазона [0, 1] 36 sdl =
SimpleDatasetLoader(preprocessors=[aap, iap]) 37 (data, labels) = sdl.load(imagePaths,
verbose =500) 38 данных = data.astype("с плавающей запятой") / 255.0
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
44
45 # преобразовать метки из целых чисел в векторы 46 trainY
= LabelBinarizer().fit_transform(trainY) 47 testY =
LabelBinarizer().fit_transform(testY)

```

Наш следующий блок кода очень важен, так как он инициализирует наш ImageDataGenerator:

```

49 # построить генератор изображений для увеличения данных 50 aug
= ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
51     height_shift_range = 0.1, shear_range = 0.2, zoom_range = 0.2, horizontal_flip
52     = True, fill_mode = «ближайший»)

```

Здесь мы позволим изображениям быть:
1. Произвольно повернутыми на ±30 градусов
2. Сдвинутыми по горизонтали и вертикали на коэффициент 0,2
3. Сдвинутыми на 0,2
4. Масштабированными путем равномерной выборки в диапазоне [0,8, 1,2]
5. Произвольное отражение по горизонтали.

В зависимости от вашего точного набора данных вы захотите настроить эти значения увеличения данных. Типично видеть диапазоны вращения между [10,30] в зависимости от вашего приложения. Горизонтальные и вертикальные сдвиги обычно попадают в диапазон [0,1,0,2] (то же самое касается значений масштабирования). Если горизонтальное отражение вашего изображения не изменяет метку класса, вы всегда должны также включать горизонтальное отражение.

Как и в предыдущем эксперименте, мы будем обучать MiniVGGNet с помощью оптимизатора SGD:

```

54 # инициализируем оптимизатор и модель 55
print("[INFO] компиляция модели...") 56 opt =
SGD(lr=0.05) 57 model = MiniVGGNet.build(width=64,
height=64, depth=3, классы = len (имена классов))
58
59 model.compile(loss="categorical_crossentropy", оптимизатор=opt,
метрики=["точность"])

```

Однако код, используемый для обучения нашей сети, должен немного измениться, поскольку теперь мы используем генератор изображений:

```

62 # обучаем сеть
63 print("[ИНФО] обучающая сеть...")
64 H = model.fit_generator(aug.flow(trainX, trainY, batch_size=32),
65     validation_data=(testX, testY), steps_per_epoch=len(trainX) // 32, epochs =100, verbose=1)
66

```

Вместо вызова метода модели `.fit` теперь нам нужно вызывать `.fit_generator`. Первый параметр `.fit_generator` — это `aug.flow`, наша функция увеличения данных, используемая для создания новых обучающих выборок из обучающих данных. `aug.flow` требует, чтобы мы передали наши обучающие данные и соответствующие метки. Нам также необходимо указать размер пакета, чтобы генератор мог создавать соответствующие пакеты при обучении сети.

Затем мы предоставляем `validation_data` в виде двух кортежей (`testX, testY`) — эти данные используются для проверки в конце каждой эпохи. Параметр `steps_per_epoch` управляет количеством пакетов в эпоху — мы можем программно определить правильное значение `steps_per_epoch`, разделив общее количество обучающих выборок на размер нашего пакета и приведя его к целому числу. Наконец, эпохи контролируют общее количество эпох, для которых наша сеть должна быть обучена, в данном случае 100 эпох.

После обучения нашей сети мы оценим ее и построим соответствующий график точности/потери:

```

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     прогнозы.argmax(ось = 1), target_names = classNames))
73
74 # график потерь и точности обучения 75
plt.style.use("ggplot") 76 plt.figure() 77 plt.plot(np.arange(0,
100), H.history["loss"], label= "train_loss") 78
plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss") 79 plt.plot(np.arange(0, 100), H.history
["acc"], label="train_acc") 80 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc") 81 plt.title("Потери при обучении и Accuracy") 82 plt.xlabel("Эпоха #") 83 plt.ylabel("Потери/точность") 84 plt.legend()
85 plt.show()

```

Обратите внимание, что мы не применяем увеличение данных к данным проверки. Мы применяем только данные дополнение к тренировочному набору.

Чтобы обучить MiniVGGNet на Цветах-17 с аугментацией данных, достаточно выполнить следующее команда:

```
$ python minivggnet_flowers17_data_aug.py --dataset ./flowers17/images [INFO] загрузка изображений...
```

```

[INFO] обработано 500/1360 [INFO]
обработано 1000/1360 [INFO]
компиляция модели...
[INFO] тренировочная сеть...
Эпоха 1/100 3 с
- потеря: 3,4073 - акк: 0,2108 - знач_потеря: 3,0306 - знач_акк: 0,1882
...
Эпоха 100/100 3 с
- потеря: 0,2769 - акк: 0,9078 - val_loss: 1,3560 - val_acc: 0,6794 [INFO] оценка сети...

```

	точность	вспомнить поддержку	f1-score
колокольчик	0,67	0,44	0,53

лютик мать-и-мачеха	0,59	0,80	0,68	20
первоцвет	0,45	0,50	0,47	18
крокус	0,82	0,45	0,58	20
нарцисс	0,67	0,30	0,41	27
	1,00	0,95	0,97	20
ромашка одуванчик	0,63	0,96	0,76	23
рябчатый ирис	0,94	0,77	0,85	22
	0,78	0,86	0,82	21
lilyvalley	0,44	0,73	0,55	15
	1,00	0,74	0,85	19
анютины	0,54	0,65	0,59	23
глазки	1,00	0,96	0,98	23
подснежник	0,80	0,84	0,82	19
	0,22	0,25	0,24	16
подсолнух тигровая лилия	0,71	0,71	0,76	16
среднее / общее	0,71	0,68	0,68	340

Сразу после того, как сеть закончит обучение, вы заметите увеличение точности с 64% до 71%, что на 10,9% больше, чем в предыдущем прогоне. Однако точность — это еще не все. реальный вопрос заключается в том, помогло ли увеличение данных предотвратить переоснащение. Чтобы ответить на этот вопрос, нам нужно изучить график потерь и точности на рис. 2.6.



Рисунок 2.6: Применение MiniVGGNet к Flowers-17 с увеличением данных. Переоснащение по-прежнему беспокойство; однако мы можем получить значительно более высокую точность классификации и меньшие потери.

Несмотря на то, что переоснащение все еще происходит, эффект значительно снижается за счет использования данных. увеличение. Опять же, имейте в виду, что эти два эксперимента идентичны — единственные изменения мы сделали, было ли применено увеличение данных. В качестве регуляризатора вы также можете увидеть

2.4 Резюме

увеличение данных оказывает влияние. Мы смогли повысить точность проверки, тем самым улучшив обобщаемость нашей модели, несмотря на снижение точности обучения.

Дополнительную точность можно получить, уменьшая скорость обучения с течением времени. Скорость обучения была специально исключена из этой главы, чтобы мы могли сосредоточиться исключительно на увеличении данных о воздействии, которое оказывает регуляризатор при обучении сверточных нейронных сетей.

2.4 Резюме

Увеличение данных — это тип метода регуляризации, который работает с обучающими данными. Как следует из названия, аугментация данных случайным образом искажает наши обучающие данные, применяя серию случайных перемещений, вращений, сдвигов и переворотов. Применение этих простых преобразований не изменяет метку класса входного изображения; однако каждое дополненное изображение можно считать «новым» изображением , которое обучающий алгоритм еще не видел. Поэтому наш обучающий алгоритм постоянно пополняется новыми обучающими образцами, что позволяет ему изучать более надежные и различительные шаблоны.

Как показали наши результаты, применение аугментации данных повысило точность нашей классификации , помогая смягчить последствия переобучения. Кроме того, увеличение данных также позволило нам обучить сверточную нейронную сеть только на 60 образцах на класс, что намного меньше рекомендуемых 1000-5000 образцов на класс.

Хотя всегда лучше собирать «естественные» обучающие выборки, в крайнем случае можно использовать аугментацию данных для преодоления ограничений небольших наборов данных. Когда дело доходит до ваших собственных экспериментов, вы должны применять дополнение данных почти к каждому эксперименту, который вы проводите. Вы должны принять небольшое снижение производительности из-за того, что ЦП теперь отвечает за случайное преобразование ваших входных данных; однако этот удар по производительности смягчается за счет использования многопоточности и дополнения ваших данных в фоновом режиме до того, как они будут переданы в поток, отвечающий за обучение вашей сети.

Опять же, почти во всех оставшихся главах пакетов Practitioner Bundle и ImageNet Bundle мы будем использовать аугментацию данных. Потратьте время, чтобы ознакомиться с этой техникой сейчас, так как она поможет вам быстрее получить более эффективные модели глубокого обучения (с использованием меньшего количества данных).

3. Сети как средства извлечения признаков

В нескольких новых главах мы обсудим концепцию трансферного обучения, возможность использовать предварительно обученную модель в качестве «ярлыка» для изучения закономерностей на данных, на которых она изначально не обучалась.

Рассмотрим традиционный сценарий машинного обучения, в котором перед нами стоят две задачи классификации. В первой задаче наша цель — обучить сверточную нейронную сеть распознавать собак и кошек на изображении (как мы сделаем в главе 10).

Затем, во втором проекте, нам нужно распознать три отдельных вида медведей: медведей гризли, белых медведей и гигантских панд. Используя стандартные методы машинного обучения, нейронных сетей и глубокого обучения, мы рассматриваем эти проблемы как две отдельные проблемы. Сначала мы соберем достаточный набор размеченных данных о собаках и кошках, а затем обучим модель на наборе данных. Затем мы повторяли процесс во второй раз, только на этот раз, собирая изображения наших пород медведей, а затем обучая модель поверх помеченного набора данных о медведях.

Трансферное обучение предлагает другую парадигму обучения — что, если бы мы могли использовать существующий предварительно обученный классификатор и использовать его в качестве отправной точки для новой задачи классификации? В контексте предложенных выше задач мы сначала обучим сверточную нейронную сеть распознавать собак и кошек. Затем мы использовали бы ту же CNN, обученную на данных о собаках и кошках, чтобы различать классы медведей, даже если данные о медведях не смешивались с данными о собаках и кошках.

Это звучит слишком хорошо, чтобы быть правдой? На самом деле это не так. Глубокие нейронные сети, обученные на крупномасштабных наборах данных, таких как ImageNet, отлично справляются с задачей трансферного обучения. Эти сети изучают набор богатых отличительных признаков для распознавания 1000 отдельных классов объектов. Имеет смысл, что эти фильтры можно повторно использовать для задач классификации, отличных от того, на чем изначально обучалась CNN.

В целом существует два типа трансферного обучения применительно к глубокому обучению для компьютерного зрения:

1. Отношение к сетям как к экстракторам произвольных признаков.
2. Удаление полностью связанных слоев существующей сети, размещение нового набора слоев FC поверх CNN и точная настройка этих весов (и, возможно, предыдущих слоев) для распознавания классов объектов.

В этой главе мы сосредоточимся в первую очередь на первом методе трансферного обучения, рассматривая

сети как средства извлечения признаков. Затем мы обсудим, как точно настроить веса сети для конкретной задачи классификации в главе 5.

3.1 Извлечение признаков с помощью предварительно обученной CNN

До этого момента мы рассматривали сверточные нейронные сети как сквозные классификаторы изображений:

1. Вводим изображение в сеть.
2. Изображение пересыпается по сети.
3. Получаем окончательные вероятности классификации с конца сети.

Однако не существует «правила», согласно которому мы должны разрешить распространение изображения по всей сети. Вместо этого мы можем остановить распространение на произвольном уровне, таком как уровень активации или объединения, извлечь значения из сети в это время, а затем использовать их в качестве векторов признаков. Например, рассмотрим сетевую архитектуру VGG16 Симоняна и Зиссермана [11] (рис. 3.1, слева).

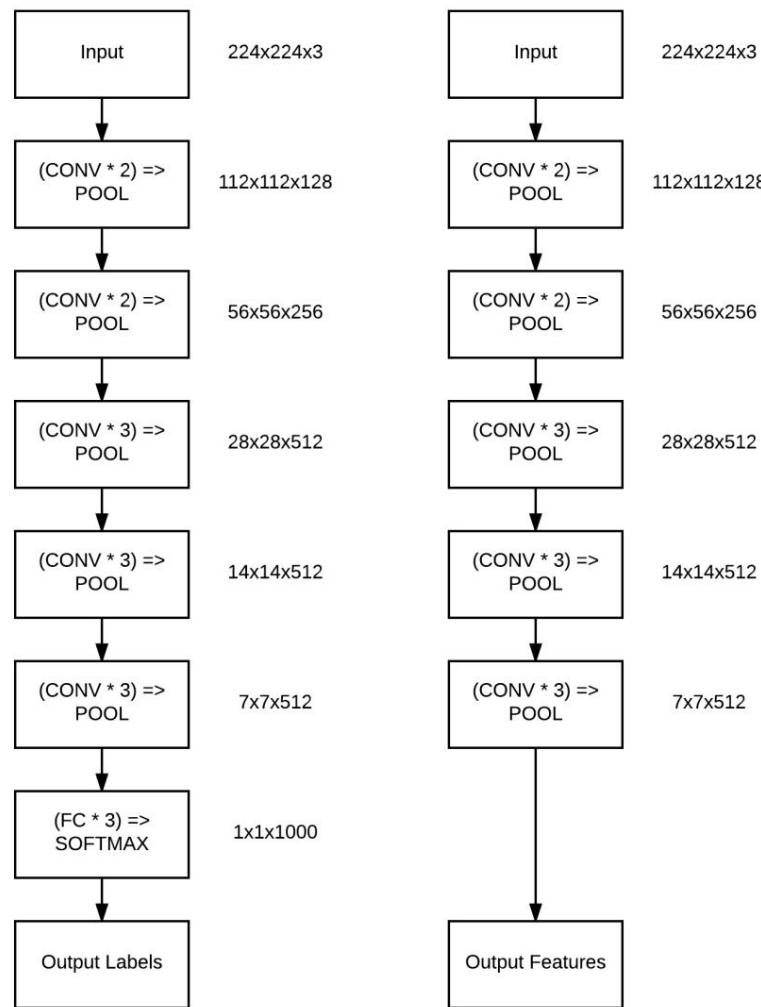


Рисунок 3.1: Слева: исходная архитектура сети VGG16, которая выводит вероятности для каждой из 1000 меток классов ImageNet. Справа: удаление слоев FC из VGG16 и возврат выходных данных последнего слоя POOL. Этот вывод будет служить нашими извлеченными функциями.

Наряду со слоями в сети мы также включили входные и выходные формы сети.

объемы для каждого слоя. Рассматривая сети как средство извлечения признаков, мы, по сути, «отсекаем» сеть в произвольной точке (обычно до полно связных слоев, но это действительно зависит от вашего конкретного набора данных).

Теперь последний слой в нашей сети — это слой с максимальным пулом (рис. 3.1, справа), который будет иметь выходную форму $7 \times 7 \times 512$, что означает наличие 512 фильтров размером 7×7 каждый. Если бы мы передавали изображение через эту сеть с удаленной головкой FC, у нас осталось бы 512 активаций 7×7 , которые либо активировались, либо не активировались в зависимости от содержимого изображения. Следовательно, мы можем взять эти $7 \times 7 \times 512 = 25\,088$ значений и рассматривать их как вектор признаков, который количественно определяет содержимое изображения.

Если мы повторим этот процесс для всего набора данных изображений (включая наборы данных, на которых VGG16 не обучался), у нас останется матрица проектирования из N изображений, каждое из которых имеет 25 088 столбцов, используемых для количественной оценки их содержимого (т. е. векторов признаков) . Учитывая наши векторы функций, мы можем обучить готовую модель машинного обучения, такую как линейный SVM, классификатор логистической регрессии или случайный лес, поверх этих функций, чтобы получить классификатор, который распознает новые классы изображений.

Имейте в виду, что сама CNN не способна распознавать эти новые классы — вместо этого мы используем CNN в качестве промежуточного экстрактора признаков. Нижестоящий классификатор машинного обучения позаботится об изучении базовых моделей функций, извлеченных из CNN.

Позже в этой главе я покажу, как вы можете использовать предварительно обученные CNN (в частности, VGG16) и библиотеку Keras для получения точности классификации > 95% для наборов данных изображений, таких как Animals, CALTECH-101 и Flowers-17. Ни один из этих наборов данных не содержит изображений, на которых обучался VGG16, но, применяя трансферное обучение, мы можем без особых усилий создавать сверхточные классификаторы изображений. Хитрость заключается в извлечении этих функций и их эффективном сохранении . Для выполнения этой задачи нам понадобится HDF5.

3.1.1 Что такое HDF5?

HDF5 — это формат двоичных данных, созданный группой HDF5 [12] для хранения гигантских числовых наборов данных на диске (слишком больших, чтобы хранить их в памяти), при этом облегчая доступ и вычисления в строках наборов данных. Данные в HDF5 хранятся иерархически, подобно тому, как файловая система хранит данные. Данные сначала определяются в группах, где группа представляет собой структуру, подобную контейнеру, которая может содержать наборы данных и другие группы. После определения группы в ней можно создать набор данных. Набор данных можно рассматривать как многомерный массив (т. е. массив NumPy) однородного типа данных (целое число, число с плавающей запятой, юникод и т. д.). Пример файла HDF5, содержащего группу с несколькими наборами данных, показан на рис. 3.2.

HDF5 написан на C; однако, используя модуль [h5py \(h5py.org\)](http://h5py.org), мы можем получить доступ к базовому API, используя язык программирования Python. Что делает h5py таким замечательным, так это простота взаимодействия с данными. Мы можем хранить огромные объемы данных в нашем наборе данных HDF5 и манипулировать данными в стиле NumPy. Например, мы можем использовать стандартный синтаксис Python для доступа и разделения строк из многотерабайтных наборов данных, хранящихся на диске, как если бы они были простыми массивами NumPy, загруженными в память. Благодаря специализированным структурам данных эти срезы и доступ к строкам выполняются молниеносно. При использовании HDF5 с h5py вы можете думать о своих данных как о гигантском массиве NumPy , который слишком велик, чтобы поместиться в основную память, но по-прежнему может быть доступен и обработан.

Возможно, лучше всего то, что формат HDF5 стандартизирован, а это означает, что наборы данных, хранящиеся в формате HDF5, изначально переносимы и могут быть доступны другим разработчикам, использующим разные языки программирования, такие как C, MATLAB и Java.

В оставшейся части этой главы мы напишем собственный класс Python, который позволит нам эффективно принимать входные данные и записывать их в набор данных HDF5. Затем этот класс будет служить двум целям:

1. Облегчить нам метод применения трансферного обучения, взяв наши извлеченные функции из VGG16 и эффективно записав их в набор данных HDF5.
2. Позвольте нам генерировать наборы данных HDF5 из необработанных изображений, чтобы ускорить обучение (глава 9).

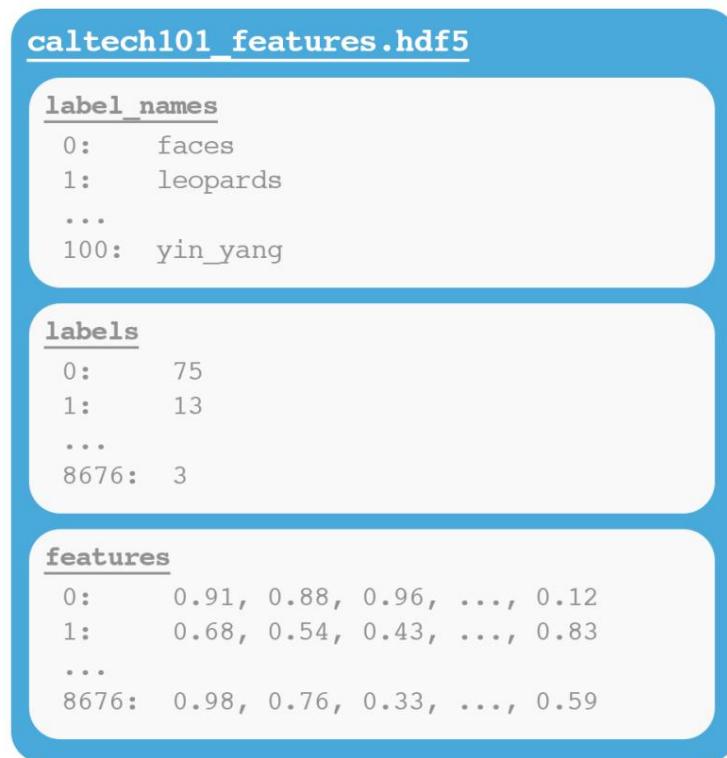


Рисунок 3.2: Пример файла HDF5 с тремя наборами данных. Первый набор данных содержит `label_names` для CALTECH-101. Затем у нас есть метки, которые сопоставляют каждое изображение с его соответствующий ярлык класса. Наконец, набор данных признаков содержит количественные оценки изображений , извлеченные CNN.

Если в вашей системе еще не установлены HDF5 и h5py , см. дополнение тарный материал к Главе 6 Starter Bundle для получения инструкций по настройке вашей системы.

3.1.2 Запись функций в набор данных HDF5

Прежде чем мы сможем даже подумать о том, чтобы рассматривать VGG16 (или любую другую CNN) как экстрактор признаков, мы сначала нужно немного развить инфраструктуру. В частности, нам нужно определить класс Python с именем `HDF5DatasetWriter`, который, как следует из названия, отвечает за получение входного набора Массивы NumPy (будь то функции, необработанные изображения и т. д.) и запись их в формат HDF5. Для этого создайте новый подмодуль в пакете pyimagesearch с именем `io` , а затем поместите файл с именем `hdf5datasetwriter.py` внутри `io`:

```

--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ИО
| | |--- __init__.py
| | |--- hdf5datasetwriter.py
| |--- НН
| |--- предварительная обработка
| |--- утилиты

```

Оттуда откройте hdf5datasetwriter.py, и мы приступим к работе:

```
1 # импортируем необходимые пакеты
2 import h5py
3 import os
```

Мы начнем легко с нашего импорта. Нам нужны только два пакета Python для создания функциональности внутри этого класса — встроенный модуль `os` и `h5py`, поэтому у нас есть доступ к HDF5. привязки.

Оттуда давайте определим конструктор:

5 класс `HDF5DatasetWriter`:

```
6     def __init__(self, dims, outputPath, dataKey="images",
7                  bufferSize=1000):
8
9         # проверяем, существует ли выходной путь, и если да, то повышаем
10        # исключение
11        если os.path.exists(outputPath):
12            поднять ValueError("Предоставленный 'outputPath' уже
13                "существует и не может быть перезаписан. Вручную удалите "файл
14                "перед продолжением", outputPath)
15
16        # открыть базу данных HDF5 для записи и создать два набора данных:
17        # один для хранения изображений/функций, а другой для хранения
18        # ярлыки классов
19        self.db = h5py.File(outputPath, "w")
20        self.data = self.db.create_dataset(dataKey, тусклый,
21                                         dtype="плавающий")
22        self.labels = self.db.create_dataset(«метки», (тусклые [0]),
23                                         dtype = "инт")
24
25        # сохранить размер буфера, затем инициализировать сам буфер
26        # вместе с индексом в наборы данных
27        self.bufSize = размер буфера
28        self.buffer = {"данные": [], "метки": []}
29        self.idx = 0
```

Конструктор `HDF5DatasetWriter` принимает четыре параметра, два из которых являются необязательными. Параметр `dims` управляет размером или формой данных, которые мы будем хранить в наборе данных. Думайте о `dims` как о `.shape` массива NumPy. Если бы мы сохраняли (сплющенный) необработанный пиксель интенсивности набора данных $28 \times 28 = 784$ MNIST, затем `dims = (70000, 784)`, поскольку имеется 70 000 примеры в MNIST, каждый с размерностью 784. Если бы мы хотели хранить необработанный CIFAR-10 изображений, то мы бы установили `dims=(60000, 32, 32, 3)`, так как всего 60 000 изображений в набор данных CIFAR-10, каждый из которых представлен RGB-изображением $32 \times 32 \times 3$.

В контексте трансферного обучения и извлечения признаков мы будем использовать архитектуру VGG16. и получение выходных данных после последнего слоя POOL . Выход последнего слоя POOL составляет $512 \times 7 \times 7$, который при сглаживании дает вектор признаков длиной 25 088. Поэтому при использовании VGG16 для извлечения признаков мы установим `dims=(N, 25088)`, где `N` — общее количество изображений в нашем набор данных.

Следующим параметром конструктора `HDF5DatasetWriter` является `outputPath` — это путь к тому месту, где наш выходной файл HDF5 будет храниться на диске. Необязательный `dataKey` — это имя набор данных, в котором будут храниться данные, из которых будет учиться наш алгоритм. По умолчанию это значение равно «изображения»,

поскольку в большинстве случаев мы будем хранить необработанные изображения в формате HDF5. Однако для этого примера, когда мы создаем экземпляр `HDF5DatasetWriter`, мы установим `dataKey="features"`, чтобы указать, что мы сохранение функций, извлеченных из CNN, в файле.

Наконец, `bufSize` управляет размером нашего буфера в памяти, который по умолчанию равен 1000. векторы/изображения. Как только мы достигнем `bufSize`, мы сбросим буфер в набор данных HDF5.

Строки 10-13 затем проверяют, существует ли уже `outputPath`. Если это так, мы поднимаем ошибку конечному пользователю (поскольку мы не хотим перезаписывать существующую базу данных).

Строка 18 открывает файл HDF5 для записи с использованием предоставленного `outputPath`. Строки 19 и 20 создайте набор данных с именем `dataKey` и предоставленными размерами — здесь мы будем хранить наши необработанные изображения/извлеченные функции. Строки 21 и 22 создают второй набор данных для хранения (целочисленные) метки классов для каждой записи в наборе данных, строки 25-28, затем инициализируют наши буфера.

Далее давайте рассмотрим метод добавления, используемый для добавления данных в наш буфер:

```

30     def add(self, rows, labels):
31         # добавляем строки и метки в буфер
32         self.buffer["данные"].extend(строки)
33         self.buffer["метки"].extend(метки)
34
35         # проверяем, нужно ли сбрасывать буфер на диск
36         если len(self.buffer["данные"]) >= self.bufSize:
37             self.flush()

```

Для метода `add` требуются два параметра: `строки`, которые мы будем добавлять в набор данных, а также с соответствующими метками классов. И `строки`, и метки добавляются в соответствующие буфера в строках 32 и 33. Если буфер заполняется, мы вызываем метод `flush` для записи буферов в файл и сбросить их.

Говоря о методе сброса, давайте теперь определим функцию:

```

39     деф флеш (сам):
40         # записываем буфера на диск, затем сбрасываем буфер
41         я = self.idx + len(self.buffer["данные"])
42         self.data[self.idx:i] = self.buffer["данные"]
43         self.labels[self.idx:i] = self.buffer["метки"]
44         self.idx = я
45         self.buffer = {"данные": [], "метки": []}

```

Если мы думаем о нашем наборе данных HDF5 как о большом массиве NumPy, нам нужно отслеживать текущий индексировать в следующую доступную строку, где мы можем хранить данные (без перезаписи существующих данных) — Страна 41 определяет следующую доступную строку в матрице. Строки 42 и 43 затем применяют массив NumPy нарезка для хранения данных и меток в буферах. Затем строка 45 сбрасывает буфера.

Мы также определим удобную служебную функцию с именем `storeClassLabels`, которая при вызове сохраните необработанные строковые имена меток классов в отдельном наборе данных:

```

47     деф storeClassLabels (я, classLabels):
48         # создайте набор данных для хранения фактических имен меток классов,
49         # затем сохраним метки классов
50         dt = h5py.special_dtype(vlen=unicode)
51         labelSet = self.db.create_dataset("label_names",
52                                         (len(classLabels),), dtype=dt)
53         labelSet[:] = классЯрлыки

```

Наконец, наша последняя функция close будет использоваться для записи любых данных, оставшихся в буферах, в HDF5 как а также закрыть набор данных:

```

55     защита близко (я):
56         # проверить, нет ли в буфере других записей, # которые нужно сбросить на
57         # диск
58         если len(self.buffer["данные"]) > 0:
59             self.flush()
60
61         # закрыть набор данных
62         self.db.close()

```

Как видите, HDF5DatasetWriter вообще не имеет ничего общего с машинным или глубоким обучением — это просто класс, который помогает нам хранить данные в формате HDF5. По мере того, как вы продолжите свою карьеру в области глубокого обучения, вы заметите, что большая часть первоначальной работы при постановке новой задачи заключается в преобразовании данных в формат, с которым вы можете работать. Когда у вас есть данные в формате , которым легко манипулировать, становится значительно проще применять методы машинного обучения и глубокого обучения к вашим данным.

При всем при этом, поскольку класс HDF5DatasetWriter является служебным классом, не относящимся к глубокому обучению и компьютерному зрению, я сделал объяснение этого класса короче, чем другие примеры кода в этой книге. Если вы обнаружите, что изо всех сил пытаетесь понять этот класс, я бы посоветовал вам:

1. Завершите чтение оставшейся части этой главы, чтобы понять, как мы используем ее в контексте извлечения признаков.
2. Потратьте время на изучение некоторых основных парадигм программирования на Python. Я привожу список рекомендуемых источников программирования на Python здесь: <http://pyimg.co/ida57>.
3. Разберите этот класс и реализуйте вручную, по частям, пока не поймете, что это такое.

Теперь, когда наш HDF5DatasetWriter реализован, мы можем перейти к фактическому извлечению функции с использованием предварительно обученных сверточных нейронных сетей.

3.2 Процесс извлечения признаков

Давайте определим скрипт Python, который можно использовать для извлечения признаков из произвольного набора данных изображения (при условии, что входной набор данных соответствует определенной структуре каталогов). Откройте новый файл, назовите его extract_features.py, и мы приступим к работе:

```

1 # импортируем необходимые пакеты 2 из
keras.applications import VGG16 3 из keras.applications
import imagenet_utils 4 из keras.preprocessing.image import
img_to_array 5 из keras.preprocessing.image import load_img 6 из
sklearn.preprocessing import LabelEncoder 7 из pyimagesearch.io
импортировать HDF5DatasetWriter 8 из путей импорта imutils 9
импортировать numpy как np 10 импортировать индикатор
выполнения 11 импортировать argparse 12 импортировать
случайные 13 импортировать ОС

```

Строки 2-13 импортируют необходимые пакеты Python. Обратите внимание, как в строке 2 мы импортируем реализацию Keras предварительно обученной сети VGG16 — это архитектура, которую мы будем использовать в качестве средства извлечения признаков. Класс LabelEncoder в строке 6 будет использоваться для преобразования меток нашего класса . от строк к целым числам. Мы также импортируем наш HDF5DatasetWriter в строку 7 , чтобы мы могли записывать функции, извлеченные из нашей CNN, в набор данных HDF5.

Один импорт, который вы еще не видели, это индикатор выполнения в строке 10. Этот пакет не имеет ничего общего с глубоким обучением, но мне нравится использовать его для длительных задач, поскольку он отображает хорошо отформатированный индикатор выполнения на вашем терминале, а также предоставляет приблизительное время, когда ваш скрипт завершит выполнение:

1 Извлечение функций 30% #####

| Расчетное время прибытия: 0:00:18

Если в вашей системе еще не установлен индикатор выполнения, вы можете установить его через:

\$ pip установить прогрессбар

В противном случае вы можете просто закомментировать все строки, которые используют индикатор выполнения (в конце концов, пакет используется только для развлечения).

Давайте перейдем к нашим аргументам 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 входному набору данных")
19 ap.add_argument("-o", "--output", required=True,
20     help="путь к выходному файлу HDF5") 21
ap.add_argument("-b", "--batch-size", type=int, default=32,
22     help="размер пакета изображений для передачи по сети") 23 ap.add_argument("-
s", "--buffer-size", type=int, default=1000, help="размер буфера извлечения объектов") 25
24     аргументов = переменные (ap.parse_args())
```

Для нашего сценария extract_features.py потребуются два аргумента командной строки, за которыми следуют два необязательных. Переключатель --dataset управляет путем к нашему входному каталогу изображений, из которых мы хотим извлечь функции. Переключатель --output определяет путь к нашему выходному файлу данных HDF5.

Затем мы можем указать --batch-size — это количество изображений в пакете, которые будут передаваться через VGG16 за раз. Здесь разумно значение 32, но вы можете увеличить его, если на вашем компьютере достаточно памяти. Переключатель --buffer-size управляет количеством извлеченных функций, которые мы будем хранить в памяти перед записью буфера для нашего набора данных HDF5. Опять же, если на вашем компьютере достаточно памяти, вы можете увеличить размер буфера.

Следующий шаг — взять пути к изображениям с диска, перемешать их и закодировать метки:

```
27 # сохранить размер партии в удобной переменной
28 бит = аргументы["размер_пакета"]
29
30 # возьмите список изображений, которые мы будем описывать, а затем случайным образом
31 # перетасуйте их, чтобы упростить обучение и тестирование сплитов с помощью 32 #
нарезки массива во время обучения
```

```

33 print("[INFO] загрузка изображений...") 34
imagePaths = list(paths.list_images(args["набор данных"])) 35
random.shuffle(imagePaths)
36
37 # извлечь метки классов из путей к изображениям, затем закодировать 38 #
метки
39 меток = [p.split(os.path.sep)[-2] для p в imagePaths] 40 le = LabelEncoder()
41 метка = le.fit_transform(labels)

```

В строке 34 мы получаем наши `imagePaths`, имена файлов для всех изображений в нашем наборе данных. Затем мы намеренно перемешиваем их в строке 35. Зачем нам это перемешивание? Что ж, имейте в виду, что в предыдущих примерах этой книги мы вычисляли разделение обучения и тестирования до обучения нашего классификатора. Однако, поскольку мы будем работать с наборами данных, которые слишком велики, чтобы поместиться в память, мы не сможем выполнить это перемешивание в памяти, поэтому мы перемешиваем пути изображений перед извлечением признаков. Затем, во время обучения, мы можем вычислить 75-процентный индекс в наборе данных HDF5 и использовать этот индекс в качестве конца обучающих данных и начала тестовых данных (этот момент станет более ясным в Разделе 3.3 ниже).

Строка 39 затем извлекает имена меток классов из наших путей к файлам, предполагая, что наши пути к файлам имеют структуру каталогов:

имя_набора_данных/{метка_класса}/example.jpg

При условии, что наш набор данных следует этой структуре каталогов (как и все примеры в этой книге), строка 39 разбивает путь на массив на основе разделителя пути ('/' на машинах Unix и '\' на Windows), а затем захватывает предпоследнюю запись в массиве — эта операция дает метку класса конкретного изображения. Получив метки, мы затем кодируем их как целые числа в строках 40 и 41 (в процессе обучения мы будем выполнять однократное кодирование).

Теперь мы можем загрузить веса сети VGG16 и создать экземпляр нашего `HDF5DatasetWriter`:

```

43 # загрузить сеть VGG16
44 print("[ИНФО] загрузка сети...") 45 model =
VGG16(weights="imagenet", include_top=False)
46
47 # инициализировать средство записи набора данных HDF5, затем сохранить метку класса
48 # имен в наборе данных
49 набор данных = HDF5DatasetWriter((len(imagePaths), 512 * 7 * 7),
50                                     args["output"], dataKey="features", bufSize=args["buffer_size"]) 51
dataset.storeClassLabels(le.classes_)

```

Строка 45 загружаем предварительно обученную сеть VGG16 с диска; однако обратите внимание, как мы включили параметр `include_top=False` — предоставление этого значения указывает на то, что окончательные полно связанные слои не должны включаться в архитектуру. Следовательно, при прямом распространении изображения по сети мы получим значения признаков после последнего слоя POOL, а не вероятности, созданные классификатором softmax в слоях FC.

Строки 49 и 50 создают экземпляр `HDF5DatasetWriter`. Первый параметр — это наши размеры набора данных, где будет всего `len(imagePaths)` изображений, каждое с вектором признаков размером $512 \times 7 \times 7 = 25\,088$. Затем в строке 51 сохраняются строковые имена меток классов в соответствии с кодировщиком меток.

Теперь пришло время выполнить фактическое извлечение признаков:

```

53 # инициализировать индикатор выполнения
54 widgets = ["Извлечение признаков:", progressbar.Percentage(), " ",
55         progressbar.Bar(), " ", progressbar.ETA()]
56 pbar = progressbar.ProgressBar(maxval = len(imagePaths),
57     виджеты=виджеты).start()
58
59 # цикл по изображениям в пачках
60 для i в np.arange(0, len(imagePaths), bs):
61     # извлечь пакет изображений и меток, затем инициализировать
62     # список актуальных изображений, которые будут передаваться по сети
63     # для извлечения признаков
64     batchPaths = imagePaths[i:i + bs]
65     batchLabels = метки[i:i + bs]
66     пакетные изображения = []

```

Строки 54-57 инициализируют наш индикатор выполнения, чтобы мы могли визуализировать и оценить, как долго функция процесса экстракции будет проходить. Опять же, использование индикатора прогресса необязательно, так что не стесняйтесь комментировать эти строки вышли.

В строке 60 мы начинаем перебирать наши imagePath партиями по --batch-size. Строки 64 и 65 извлекают пути к изображениям и метки для соответствующего пакета, а Стока 66 инициализирует список для хранения изображений, которые должны быть загружены и загружены в VGG16.

Подготовка изображения для извлечения признаков точно такая же, как подготовка изображения для классификации через CNN:

```

68     # цикл по изображениям и меткам в текущем пакете
69     для (j, imagePath) в перечислении (batchPaths):
70         # загружаем входное изображение с помощью вспомогательной утилиты Keras
71         # при изменении размера изображения до 224x224 пикселей
72         изображение = load_img (imagePath, target_size = (224, 224))
73         изображение = img_to_array (изображение)
74
75         # предварительно обработать изображение путем (1) увеличения размеров и
76         # (2) вычитание средней интенсивности пикселя RGB из
77         # Набор данных ImageNet
78         изображение = np.expand_dims (изображение, ось = 0)
79         изображение = imagenet_utils.preprocess_input(изображение)
80
81         # добавляем изображение в пакет
82         batchImages.append(изображение)

```

В строке 69 мы перебираем каждый путь изображения в пакете. Каждое изображение загружается с диска и преобразован в массив, совместимый с Keras (строки 72 и 73). Затем мы предварительно обрабатываем изображение в Lines . 78 и 79 с последующим добавлением его в batchImages (строка 82).

Чтобы получить наши векторы признаков для изображений в пакетных изображениях, все, что нам нужно сделать, это вызвать метод .predict метод модели:

```

84     # передавать изображения по сети и использовать выходные данные как
85     # наши актуальные возможности
86     пакетные изображения = np.vstack (пакетные изображения)
87     функции = model.predict (batchImages, batch_size = bs)
88

```

```

89      # измените форму признаков так, чтобы каждое изображение было представлено
90      # сглаженным вектором признаков 'MaxPooling2D' outputs functions =
91      features.reshape((features.shape[0], 512 * 7 * 7))
92
93      # добавляем функции и метки в наш набор данных HDF5
94      dataset.add(features, batchLabels) pbar.update(i)
95

```

Мы используем метод `.vstack` NumPy в строках 86 , чтобы «вертикально сложить» наши изображения так, чтобы они имели форму (N, 224, 224, 3), где N — размер пакета.

Передача пакетов изображений через нашу сеть дает наши фактические векторы признаков — помните, мы отсекли полно связные слои в начале VGG16, поэтому теперь у нас остались значения после последней операции максимального объединения (строка 87). Однако выход POOL имеет форму (N, 512, 7, 7), что означает наличие 512 фильтров, каждый размером 7× 7. Чтобы рассматривать эти значения как вектор признаков, нам нужно объединить их в массив формы (N, 25088), что и делает строка 91 . Стока 94 добавляет наши функции и пакетные метки в наш набор данных HDF5.

Наш последний блок кода обрабатывает закрытие нашего набора данных HDF5:

```

97 # закрыть набор данных
98 набор данных.close() 99 pbar.finish()

```

В оставшейся части этого раздела мы собираемся попрактиковаться в извлечении признаков с помощью предварительно обученной CNN из различных наборов данных.

3.2.1 Извлечение признаков из животных

Первый набор данных, который мы собираемся извлечь с помощью VGG16, — это наш набор данных «Животные». Этот набор данных состоит из 3000 изображений трех классов: собак, кошек и панд. Чтобы использовать VGG16 для извлечения функций из этих изображений, просто выполните следующую команду:

```
$ python extract_features.py --dataset ../datasets/animals/images \ --output ../datasets/animals/
    hdf5/features.hdf5 [INFO] загрузка изображений...
```

```
[INFO] загрузка сети...
Особенности извлечения: 100% ##### | Время: 0:00:35
```

Используя свой графический процессор Titan X, я смог извлечь функции из всех 3000 изображений примерно за 35 секунд. После выполнения скрипта загляните в каталог `animals/hdf5` , и вы найдете файл с именем `features.hdf5`:

```
$ ls ../наборы данных/животные/hdf5/
особенности.hdf5
```

Чтобы изучить файл `.hdf5` для набора данных Animals, запустите оболочку Python:

```
$ python
>>> import h5py >>>
p = "../datasets/animals/hdf5/features.hdf5" >>> db = h5py.File(p) >>>
list(db.keys()) [u'особенности', u'label_names', u'метки']
```

Обратите внимание, что в нашем файле HDF5 есть три набора данных: объекты, имена_меток и метки. Набор данных функций — это место, где хранятся наши фактически извлеченные функции. Вы можете изучить форму этого набора данных, используя следующие команды:

```
>>> db["features"].shape (3000,
25088) >>> db["labels"].shape
(3000,) >>> db["label_names"].shape
(3,)
```

Обратите внимание на .shape (3000, 25088) — этот результат означает, что каждое из 3000 изображений в нашем наборе данных Animals количественно оценивается с помощью вектора признаков длиной 25 088 (т. е. значений внутри VGG16 после последней операции POOL). Позже в этой главе мы узнаем, как мы можем обучить классификатор этим функциям.

3.2.2 Извлечение признаков из CALTECH-101

Точно так же, как мы извлекли признаки из набора данных Animals, мы можем сделать то же самое с CALTECH-101:

```
$ python extract_features.py --dataset ..datasets/caltech-101/images \
hdf5/features.hdf5 [INFO] загрузка изображений...
[INFO] загрузка сети...
Особенности извлечения: 100% ##### | Время: 0:01:27
```

Теперь у нас есть файл с именем features.hdf5 в каталоге caltech-101/hdf5:

```
$ ls ..наборы данных/caltech-101/hdf5/
особенности.hdf5
```

Изучив этот файл, вы увидите, что каждое из 8 677 изображений представлено 25 088-мерным вектором признаков.

3.2.3 Извлечение признаков из цветов-17

Наконец, давайте применим извлечение признаков CNN к набору данных Flowers-17:

```
$ python extract_features.py --dataset ..datasets/flowers17/images \
hdf5/features.hdf5 [INFO] загрузка изображений...
[INFO] загрузка сети...
Особенности извлечения: 100% ##### | Время: 0:00:19
```

Изучив файл features.hdf5 для Flowers-17, вы увидите, что каждое из 1360 изображений в наборе данных количественно оценивается с помощью вектора признаков с 25 088 размерностями.

3.3 Обучение классификатора на извлеченных функциях

Теперь, когда мы использовали предварительно обученную CNN для извлечения функций из нескольких наборов данных, давайте посмотрим, насколько эти функции действительно различаются, особенно с учетом того, что VGG16 был обучен на ImageNet, а не на Animals, CALTECH-101 или Flowers-17 ..

Возьмите секунду и рискните предположить, насколько хороша простая линейная модель при использовании этих функций для классификации изображения — вы бы предположили, что точность классификации выше 60%? 70% кажутся неразумными? Неужели 80% маловероятно? И точность классификации 90% была бы непостижимой, верно?

Давайте выясним для себя. Откройте новый файл, назовите его train_model.py и вставьте следующий код:

```
1 # импортируем необходимые пакеты 2
из sklearn.linear_model import LogisticRegression 3 из
sklearn.model_selection import GridSearchCV 4 из sklearn.metrics
import classification_report 5 import argparse 6 import pickle 7 import
h5py
```

Строки 2-7 импортируют необходимые пакеты Python. Класс GridSearchCV будет использоваться, чтобы помочь нам преобразовать параметры в наш классификатор LogisticRegression . Мы будем использовать pickle для сериализации нашей модели LogisticRegression на диск после обучения. Наконец, будет использоваться h5py , чтобы мы могли взаимодействовать с нашим набором функций HDF5.

Теперь мы можем проанализировать наши аргументы командной строки:

```
9 # построить разбор аргумента и разобрать аргументы 10 ap =
argparse.ArgumentParser() 11 ap.add_argument("-d", "--db", required=True,
help="path HDF5 database")
12
13 ap.add_argument("-m", "-model", required=True,
14     help="путь к выходной модели") 15
ap.add_argument("-j", "--jobs", type=int, default=-1,
16     help="количество заданий для запуска при настройке
гиперпараметров") 17 args = vars(ap.parse_args())
```

Наш скрипт требует два аргумента командной строки, за которыми следует третий необязательный: 1. --db: путь к нашему набору данных HDF5, содержащему наши извлеченные функции и метки классов. 2. --model: здесь мы указываем путь к нашему выходному классификатору логистической регрессии. 3. --jobs: необязательное целое число, используемое для указания количества одновременных заданий при запуске поиска по сетке, чтобы настроить наши гиперпараметры на модель логистической регрессии.

Давайте откроем наш набор данных HDF5 и определим, где будет наше разделение обучения/тестирования:

```
19 # открыть базу данных HDF5 для чтения, затем определить индекс 20 # разделения
обучения и тестирования при условии, что эти данные уже были перемешаны 21 #
до записи на диск 22 db = h5py.File(args["db"], "r") 23 i = int(db["labels"].shape[0] * 0,75)
```

Как я упоминал ранее в этой главе, мы намеренно перетасовывали наши пути к изображениям перед записью связанных изображений/векторов объектов в набор данных HDF5 — причина этого становится ясной в строках 22 и 23.

Учитывая, что наш набор данных слишком велик, чтобы поместиться в память, нам нужен эффективный метод для определения разделения обучения и тестирования. Поскольку мы знаем, сколько записей содержится в наборе данных HDF5 (и мы знаем, что хотим использовать 75% данных для обучения и 25% для оценки), мы можем просто вычислить 75%-й индекс i в базе данных. Любые данные до индекса i считаются данными обучения, а все данные после i — проверочными данными.

Учитывая наши тренировочные и тестовые сплиты, давайте обучим наш классификатор логистической регрессии:

```

25 # определяем набор параметров, которые мы хотим настроить, затем запускаем поиск по сетке 26 #,
где мы оцениваем нашу модель для каждого значения C 27 print("[INFO] настройки гиперпараметров...")
28 params = {"C" : [0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]} 29 модель = GridSearchCV(LogisticRegression(),
params, cv=3,
30         n_jobs=args["jobs"]) 31
model.fit(db["features"][:i], db["labels"][:i]) 32 print("[INFO] лучшие гиперпараметры:
{}".format(модель.best_params_))
33
34 # оценить модель
35 print("[INFO] оценка...") 36 preds =
model.predict(db["features"][:i]) 37 print(classification_report(db["labels"]
[i:], preds, target_names=db["названия_меток"]))
38

```

Строки 28-31 выполняют поиск по сетке параметра C , строгости классификатора логистической регрессии, чтобы определить оптимальное значение. Полный подробный обзор логистической регрессии выходит за рамки этой книги, поэтому см. примечания Эндрю Нг для подробного обзора классификатора логистической регрессии [13].

Обратите внимание на то, как мы указываем обучающие данные и обучающие метки через срезы массива:

```

db["функции"][:i] db["ярлыки"]
[:i]

```

Опять же, любые данные до индекса i являются частью нашего обучающего набора. Как только лучшие гиперпараметры найдены, мы затем оцениваем классификатор на данных тестирования (строки 36-38).

Обратите внимание, что доступ к нашим тестовым данным и тестовым меткам осуществляется через срезы массива:

```

БД["функции"][:i] БД["метки"]
[:i]

```

Все, что находится после индекса i , является частью нашего тестового набора. Несмотря на то, что наш набор данных HDF5 находится на диске (и слишком велик, чтобы поместиться в память), мы все равно можем обращаться с ним, как если бы это был массив NumPy, что является одним из огромных преимуществ совместного использования HDF5 и h5py для глубокого обучения и машинного обучения . учебные задачи.

Наконец, мы сохраняем нашу модель LogisticRegression на диск и закрываем базу данных:

```

40 # сериализовать модель на диск
41 print("[INFO] сохранение модели...")

```

```

42 f = открыть(args["модель"], "wb")
43 f.write(pickle.dumps(model.best_estimator_))
44 ф.закрыть()
45
46 # закрыть базу
47 дБ.close()

```

Также обратите внимание, что нет специального кода, связанного ни с животными, ни с CALTECH-101, ни с наборами данных Flowers-17 — если наш входной набор данных изображений соответствует структуре каталогов. Подробно описано в разделе 3.2 выше, мы можем использовать как `extract_features.py`, так и `train_model.py` для быстро создавать надежные классификаторы изображений на основе функций, извлеченных из CNN. Какой ты крепкий спросить? Пусть говорят результаты.

3.3.1 Результаты на животных

Чтобы обучить классификатор логистической регрессии на функциях, извлеченных через сеть VGG16 на Набор данных животных, просто выполните следующую команду:

```

$ python train_model.py --db ..datasets/animals/hdf5/features.hdf5 \
--model animals.cpickle
[INFO] настройка гиперпараметров...
[INFO] лучшие гиперпараметры: {'C': 0.1}
[INFO] оценка...

      точность      вспомнить поддержку f1-score

кошки       0,96       0,98       0,97       252
             0,98       0,95       0,97       253
собаки панда 0,98       1,00       0,99       245
 среднее / общее 0,98       0,98       0,98       750

```

Обратите внимание, что мы можем достичь точности классификации 98% ! Это число является огромным и может служить доказательством от нашего предыдущего лучшего 71% в главе 12 Starter Bundle.

3.3.2 Результаты CALTECH-101

Эти невероятные результаты сохраняются и в наборе данных CALTECH-101. Выполните эту команду, чтобы оценить производительность функций VGG16 на CALTECH-101:

```

$ Python train_model.py \
--db ..наборы данных /caltech-101/hdf5/features.hdf5 \
--модель caltech101.cpickle
[INFO] настройка гиперпараметров...
[INFO] лучшие гиперпараметры: {'C': 1000.0}
[INFO] оценка...

      точность      вспомнить поддержку f1-score

Лица        1,00       0,98       0,99       114
Faces_easy   0,98       1,00       0,99       104
Леопарды    1,00       1,00       1,00        44
Мотоциклы   1,00       1,00       1,00       197
...
виндзор_стул 0,92       0,92       0,92        13

```

гаечный ключ	0,88	0,78	0,82	9
Инь Янь	1,00	1,00	1,00	11
среднее / общее	0,96	0,96	0,96	2170

На этот раз мы можем получить точность классификации 96% по 101 отдельной категории объектов. с минимальными усилиями!

3.3.3 Результаты по цветам-17

Наконец, давайте применим функции VGG16 к набору данных Flowers-17, где ранее мы изо всех сил пытались сломайте 71-процентную точность даже при использовании увеличения данных:

	точность	вспомнить поддержку	f1-score	
колокольчик	1,00	1,00	1,00	25
лютик мать-и-мачеха	0,90	0,78	0,84	23
первоцвет	0,67	0,95	0,78	19
крокус	0,94	1,00	0,97	16
нарцисс	0,94	0,77	0,85	22
	1,00	0,95	0,97	20
ромашка одуванчик	1,00	1,00	1,00	18
рябчатый ирис	1,00	0,96	0,98	23
	1,00	0,94	0,97	16
lilyvalley	0,73	0,94	0,82	17
	0,95	1,00	0,98	20
анютины	0,95	0,72	0,82	29
глазки	0,96	1,00	0,98	24
подснежник	1,00	1,00	1,00	12
	0,76	0,84	0,80	19
подсолнух тигровая лилия	0,94	0,94	0,97	17
подсолнух тигровая лилия	0,94	0,94	0,97	17
среднее / общее	0,93	0,92	0,92	340

На этот раз мы достигаем точности классификации 93%, что намного выше, чем 71% раньше. Очевидно, сети, такие как VGG, способны выполнять трансферное обучение, кодируя свои различительные функции в выходных активациях, которые мы можем использовать для обучения наших собственных классификаторов изображений.

3.4 Резюме

В этой главе мы начали изучать трансферное обучение, концепцию использования предварительно обученной сверточной нейронной сети для классификации меток классов за пределами того, на чем она была обучена изначально. В целом, существует два метода передачи обучения применительно к глубокому обучению и компьютерное зрение:

1. Работайте с сетями как с экстракторами признаков, передающими изображение до заданного слоя и затем берут эти активации и рассматривают их как векторы признаков.

3.4 Резюме

2. Точная настройка сетей путем добавления совершенно нового набора полносвязных слоев в начало сети и настройки этих слоев FC для распознавания новых классов (при использовании тех же базовых фильтров CONV).

В этой главе мы сосредоточились исключительно на компоненте извлечения признаков трансферного обучения, демонстрируя, что глубокие CNN, такие как VGG, Inception и ResNet, способны действовать как мощные машины извлечения признаков, даже более мощные, чем созданные вручную алгоритмы, такие как HOG [14]., SIFT [15] и Local Binary Patterns [16], и это лишь некоторые из них. Всякий раз, когда вы подходите к новой проблеме с помощью глубокого обучения и сверточных нейронных сетей, всегда думайте, даст ли применение извлечения признаков достаточную точность — если да, вы можете полностью пропустить процесс обучения сети, сэкономив массу времени, усилий и головной боли.

Мы рассмотрим мой оптимальный путь применения методов глубокого обучения, таких как извлечение признаков, тонкая настройка и обучение с нуля, в главе 8. А пока давайте продолжим изучение трансферного обучения.

4. Понимание точности ранга 1 и ранга 5

Прежде чем мы зайдем слишком далеко в обсуждении продвинутых тем глубокого обучения (таких как трансферное обучение), давайте сначала сделаем шаг назад и обсудим концепцию точности ранга 1, ранга 5 и ранга N. Читая литературу по глубокому обучению, особенно в области компьютерного зрения и классификации изображений, вы, вероятно, столкнетесь с концепцией ранжированной точности. Например, почти все статьи, в которых представлены методы машинного обучения, оцениваемые на наборе данных ImageNet, представляют свои результаты с точки зрения точности ранга 1 и ранга 5 (позже мы узнаем, почему точность ранга 1 и ранга 5 сообщается позже) . в этой главе).

Что такое точность ранга 1 и ранга 5? И чем они отличаются от традиционной точности (т.е. прецизионности)? В этой главе мы обсудим ранжированную точность, узнаем, как ее реализовать , а затем применим ее к моделям машинного обучения, обученным на наборах данных Flowers-17 и CALTECH-101.

4.1 Оценка точности



Рисунок 4.1: Слева: исходное изображение лягушки, которое наша нейронная сеть попытается классифицировать. Справа: исходное изображение автомобиля.

Ранжированную точность лучше всего объяснить на примере. Предположим, мы оцениваем нейронную сеть, обученную на наборе данных CIFAR-10, который включает десять классов: самолет, автомобиль,

Вероятность метки класса	Вероятность метки класса
Самолет 0,0%	Самолет 1,1%
Автомобиль 0,0%	Автомобиль 38,7%
Птица 2,1%	Птица 0,0%
Кошка 0,03%	Кошка 0,5%
Олень 0,01%	Олень 0,0%
Собака 0,56%	Собака 0,4%
Лягушка 97,3%	Лягушка 0,11%
Лошадь 0,0%	Лошадь 1,4%
Судно 0,0%	Судно 2,39%
Грузовая машина 0,0%	Грузовая машина 55,4%

Таблица 4.1: Слева: вероятности меток классов, возвращенные нашей нейронной сетью для рисунка 4.1 (слева).

Справа: вероятности меток классов, возвращенные нашей сетью для рисунка 4.1 (справа).

птица, кошка, олень, собака, лягушка, лошадь, корабль и грузовик. Учитывая следующее входное изображение (рис. 4.1, слева) мы просим нашу нейронную сеть вычислить вероятности для каждой метки класса — нейронная сеть затем возвращает вероятности меток класса, перечисленные в таблице 4.1 (слева).

Меткой класса с наибольшей вероятностью является лягушка (97,3%), что действительно является правильным прогнозом. Если бы мы повторили этот процесс:

1. Шаг № 1: вычисление вероятностей меток класса для каждого входного изображения в наборе данных.
2. Шаг № 2: определение, равна ли метка истинности прогнозируемой метке класса с наибольшая вероятность.
3. Шаг № 3: подсчет количества раз, когда шаг № 2 верен.

Мы достигли бы нашей точности первого ранга. Таким образом, точность ранга 1 представляет собой процент прогнозы, в которых верхний прогноз соответствует метке достоверности — это «стандартный» тип точность, которую мы привыкли вычислять: возьмите общее количество правильных прогнозов и разделите его на количество точек данных в наборе данных.

Затем мы можем расширить эту концепцию до точности пятого ранга. Вместо того, чтобы заботиться только о количестве один прогноз, нас интересуют топ-5 прогнозов. Теперь наш процесс оценки выглядит следующим образом:

1. Шаг № 1: вычислить вероятности меток класса для каждого входного изображения в наборе данных.
2. Шаг № 2: Отсортируйте прогнозируемые вероятности меток классов в порядке убывания, чтобы метки с большей вероятностью помещаются в начало списка.
3. Шаг № 3. Определите, существует ли метка достоверности среди первых 5 меток, предсказанных на шаге № 2.
4. Шаг № 4: подсчитайте, сколько раз шаг № 3 верен.

Ранг-5 — это просто расширение точности ранга 1: вместо того, чтобы заботиться только о прогнозе №1 из классификатора мы будем учитывать топ-5 прогнозов из сети. Например, давайте снова рассмотрим входное изображение, которое должно быть отнесено к категории CIFAR-10 на основе произвольная нейронная сеть (рис. 4.1, справа). После прохождения через нашу сеть мы получаем вероятности меток класса подробно описаны в таблице 4.1 (справа).

Наше изображение явно автомобиля; однако наша сеть сообщила, что грузовик является лучшим прогнозом — это будет считаться неверным прогнозом для точности ранга 1. Но если мы изучим топ-5 прогнозов сети, мы видим, что автомобиль на самом деле является прогнозом номер два, что было бы точным при вычислении точности ранга 5. Этот подход можно легко расширить также с произвольной точностью ранга N; однако обычно мы вычисляем только ранг-1 и ранг-5 точности — в связи с чем возникает вопрос, зачем вообще вычислять точность ранга 5?

Для набора данных CIFAR-10 вычисление точности ранга 5 немного глупо, но для больших и сложных

наборы данных, особенно для мелкозернистой классификации, часто бывает полезно посмотреть на топ-5 прогнозов от данного CNN. Возможно, лучший пример того, почему мы вычисляем точность ранга 1 и ранга 5, может можно найти в Szegedy et al. [17] где мы видим сибирскую хаски слева и эскимосскую собаку справа (рис. 4.2). Большинство людей не смогли бы распознать разницу между двумя животными; однако оба этих класса являются допустимыми метками в наборе данных ImageNet.



Рисунок 4.2: Слева: сибирский хаски. Справа: эскимосская собака.

При работе с большими наборами данных, которые охватывают множество меток классов со схожими характеристиками, мы часто проверяйте точность ранга 5 как расширение точности ранга 1, чтобы увидеть, как работает наша сеть. выполнение. В идеальном мире наша точность ранга 1 будет увеличиваться с той же скоростью, что и наша точность ранга 5. ТОЧНОСТЬ, но на сложных наборах данных это не всегда так.

Поэтому мы также проверяем точность ранга 5, чтобы убедиться, что наша сеть все еще «обучается». в более поздние эпохи. Это может быть случай, когда точность ранга 1 стагнирует к концу обучения, но точность ранга 5 продолжает улучшаться по мере того, как наша сеть изучает больше отличительных признаков (но недостаточно разборчивы, чтобы превзойти лучшие прогнозы №1). Наконец, в зависимости от изображения задачу классификации (каноническим примером является ImageNet), вы должны сообщить оба ваши ранг-1 и ранг-5 точности вместе.

4.1.1 Измерение точности ранга 1 и ранга 5

Вычисление точности ранга 1 и ранга 5 может быть выполнено путем построения простой функции полезности. Внутри нашего модуля `pyimagesearch` мы добавим эту функциональность в подмодуль `utils`, добавив файл с именем `rank.py`:

```

--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ИО
| |--- НН
| |--- предварительная обработка
| |--- утилиты
| | |--- __init__.py
| | |--- captchahelper.py
| | |--- Ranked.py

```

Откройте `rank.py`, и мы определим функцию `rank5_accuracy`:

```

1 # импортируем необходимые пакеты
2 import numpy as np
3
4 def rank5_accuracy(предыдущие, метки):

```

```

5      # инициализируем точность rank-1 и rank-5
6      ранг1 = 0
7      ранг5 = 0

```

Строка 4 определяет нашу функцию rank5_accuracy. Этот метод принимает два параметра:

- **pres**: матрица $N \times T$, где N , количество строк, содержит вероятности, связанные с каждой меткой класса T .
- **labels**: основные метки для изображений в наборе данных.

Затем мы инициализируем точность ранга 1 и ранга 5 в строках 6 и 7 соответственно.

Давайте продолжим и вычислим точность ранга 1 и ранга 5:

```

9      # цикл по предсказаниям и меткам истинности
10     for (p, gt) в zip(предварительно, метки):
11         # сортируем вероятности по их индексу в порядке убывания
12         # упорядочиваем так, чтобы более увереные догадки были первыми
13         # впереди списка
14         p = np.argsort(p)[::-1]
15
16         # проверяем, входит ли метка истинности в топ-5
17         # предсказания
18         если gt в p[:5]:
19             ранг5 += 1
20
21         # проверяем, является ли истина предсказанием №1
22         если gt == p[0]:
23             ранг1 += 1

```

В строке 10 мы начинаем перебирать прогнозы и метки классов истинности для каждого примера в наборе данных. В строке 14 вероятности предсказаний p сортируются в порядке убывания, так что индексы наибольших вероятностей помещаются в начало списка. Если существует метка достоверности в топ-5 прогнозов мы увеличиваем нашу переменную $rank5$ (строки 18 и 19). Если наземная правда $label$ равна первой позиции, мы увеличиваем нашу переменную $rank1$ (строки 22 и 23).

Наш последний блок кода обрабатывает преобразование ранга 1 и ранга 5 в проценты путем деления на общее количество этикеток:

```

25     # вычислить окончательную точность ранга 1 и ранга 5
26     ранг1 /= число с плавающей запятой (длина (метки))
27     ранг5 /= число с плавающей запятой (длина (метки))
28
29     # вернуть кортеж с точностью 1 и 5 ранга
30     возврат (ранг1, ранг5)

```

Строка 30 возвращает в вызывающую функцию два кортежа с точностью ранга 1 и ранга 5.

4.1.2 Реализация ранжированной точности

Чтобы продемонстрировать, как вычислить точность ранга 1 и ранга 5 для набора данных, вернемся к главе 3, где мы использовали предварительно обученную сверточную нейронную сеть в наборе данных ImageNet в качестве функции. экстрактор. На основе этих извлеченных функций мы обучили классификатор логистической регрессии на данных. и оценил модель. Теперь мы расширим наши отчеты о точности, включив в них точность 5-го ранга.

Пока мы вычисляем точность ранга 1 и ранга 5 для нашей модели логистической регрессии, сохраняйте иметь в виду, что точность ранга 1 и ранга 5 может быть вычислена для любого машинного обучения,

сеть или модель глубокого обучения — обе эти метрики часто встречаются за пределами сообщества глубокого обучения. После всего сказанного откройте новый файл, назовите его rank_accuracy.py и вставьте следующий код:

```
1 # импортируем необходимые пакеты 2
из pyimagesearch.utils.ranked import rank5_accuracy 3 import argparse 4
import pickle 5 import h5py
```

Строки 2-5 импортируют необходимые пакеты Python. Мы будем использовать наш недавно определенный rank5_accuracy. функция для вычисления точности ранга 1 и ранга 5 наших прогнозов соответственно. Пакет pickle используется для загрузки нашего предварительно обученного классификатора scikit-learn с диска. Наконец, h5py будет использоваться для взаимодействия с нашей базой данных функций HDF5, извлеченных из нашей CNN в главе 3.

Следующим шагом будет разбор наших аргументов командной строки:

```
7 # построить разбор аргумента и разобрать аргументы 8 ap =
argparse.ArgumentParser() 9 ap.add_argument("-d", "--db", required=True,
10           help="путь к базе данных HDF5")
11 ap.add_argument("-m", "--model", required=True, help="путь к
12           предварительно обученной модели") 13 args =
vars(ap.parse_args())
```

Наш скрипт потребует два аргумента: --db, который является путем к нашей базе данных HDF5. извлеченные функции sand --model, путь к нашему предварительно обученному классификатору логистической регрессии.

Следующий блок кода обрабатывает загрузку предварительно обученной модели с диска, а также определение индекса разделения обучения и тестирования в наборе данных HDF5, предполагая, что 75% данных использовались для обучения и 25% для тестирования:

```
15 # загружаем предварительно
обученную модель 16 print("[INFO] загружаем предварительно
обученную модель...") 17 model = pickle.loads(open(args["model"], "rb").read())
18
19 # открыть базу данных HDF5 для чтения, затем определить индекс 20 # разделения
обучения и тестирования при условии, что эти данные уже были перемешаны 21 #
до записи на диск 22 db = h5py.File(args["db"], "r") 23 i = int(db["labels"].shape[0] * 0,75)
```

Наконец, давайте вычислим наши точности ранга 1 и ранга 5:

```
25 # делаем прогнозы на тестовом наборе, затем вычисляем точность ранга 1 26 # и ранга 5
```

```
27 print("[INFO] прогнозирование...") 28
preds = model.predict_proba(db["features"][:i]) 29 (rank1, rank5) =
rank5_accuracy(preds, db["метки"][:i])
30
31 # отображать точность rank-1 и rank-5 32 print("[INFO]
rank-1: {:.2f}%".format(rank1 * 100))
```

```
33 print("[INFO] rank-5: {:.2f}%".format(rank5 * 100))
34
35 # закрыть базу данных
36 db.close()
```

В строке 28 вычисляются вероятности для каждой метки класса для каждой точки данных в тестовом наборе. Основываясь на предсказанных вероятностях и метках достоверности данных тестирования, мы можем вычислить ранжированные точности в строке 29. Строки 32 и 33 затем отображают ранг 1 и ранг 5 на нашем терминале соответственно.

Обратите внимание, что мы закодировали этот пример таким образом, что он будет работать с любым примером из главы 3, где мы извлекли функции из CNN, а затем обучили модель scikit-learn поверх функций. Позже в пакетах Practitioner Bundle и ImageNet Bundle мы вычислим точность ранга 1 и ранга 5 для сверточных нейронных сетей, обученных с нуля.

4.1.3 Ранжированная Точность на Цветах-17

Для начала давайте вычислим точность ранга 1 и ранга 5 для набора данных Flowers-17:

```
$ python rank_accuracy.py --db ../datasets/flowers17/hdf5/features.hdf5 \ --model ../chapter03-
    feature_extraction/flowers17.cpickle [INFO] загружает предварительно обученную модель...
[INFO] предсказание...
[ИНФО] ранг-1: 92,06%
[ИНФО] ранг-5: 99,41%
```

В наборе данных Flowers-17 мы получаем точность ранга 1 92,06%, используя классификатор логистической регрессии, обученный на функциях, извлеченных из архитектуры VGG16. Изучая точность 5-го ранга, мы видим, что наш классификатор почти идеален, достигая 99,41% точности 5-го ранга.

4.1.4 Ранжированная точность CALTECH-101

Давайте попробуем другой пример, этот на большем наборе данных CALTECH-101:

```
$ python rank_accuracy.py --db ../datasets/caltech101/hdf5/features.hdf5 \ --model ../chapter03-
    feature_extraction/caltech101.cpickle [ИНФО] загружает предварительно обученную модель...
[INFO] предсказание...
[ИНФО] ранг-1: 95,58%
[ИНФО] ранг-5: 99,45%
```

Здесь мы получаем точность ранга 1 95,58 % и точность ранга 5 99,45 %, существенное улучшение по сравнению с предыдущими методами компьютерного зрения и машинного обучения, которые изо всех сил пытались преодолеть 60-процентную точность классификации.

4.2 Резюме

В этой главе мы рассмотрели концепцию точности ранга 1 и ранга 5. Точность ранга 1 — это количество раз, когда наша метка достоверности совпадает с меткой класса с наибольшей вероятностью. Точность ранга 5 распространяется на точность ранга 1, позволяя ей быть немного более «снисходительной» — здесь мы вычисляем точность ранга 5 как количество раз, когда наша метка достоверности появляется в топ-5 меток прогнозируемых классов с наибольшая вероятность.

4.2 Резюме

55

Обычно мы сообщаем о точности 5-го ранга для больших сложных наборов данных, таких как ImageNet, где даже людям часто трудно правильно маркировать изображение. В этом случае мы будем считать прогноз для нашей модели «правильным», если метка истинности просто существует в его пяти лучших прогнозах. Как мы обсуждали в главе 9 Starter Bundle, сеть, которая действительно хорошо обобщает, будет давать контекстуально схожие прогнозы в своих пяти лучших вероятностях.

Наконец, имейте в виду, что точность ранга 1 и ранга 5 не является специфичной для глубокого обучения и классификация изображений — вы часто будете видеть эти показатели и в других задачах классификации.

5. Тонкая настройка сетей

В главе 3 мы узнали, как обращаться с предварительно обученной сверточной нейронной сетью как с экстрактором признаков. Используя этот экстрактор функций, мы передали наш набор данных изображений по сети, извлекли активации на заданном уровне и сохранили значения на диск. Затем стандартный классификатор машинного обучения (в данном случае логистическая регрессия) был обучен поверх функций CNN, точно так же, как если бы мы использовали созданные вручную функции, такие как SIFT [15], HOG [14], LBP [16] и т. д. Этот подход к извлечению признаков CNN, называемый трансферным обучением, получил замечательную точность, намного более высокую, чем любой из наших предыдущих экспериментов с наборами данных Animals, CALTECH-101 или Flowers-17.

Но есть еще один тип трансферного обучения, который может превзойти метод извлечения признаков, если у вас достаточно данных. Этот метод называется тонкой настройкой и требует от нас выполнения «сетевой хирургии». Во-первых, мы берем скальпель и отрезаем окончательный набор полносвязных слоев (то есть «голову» сети) от предварительно обученной сверточной нейронной сети, такой как VGG, ResNet или Inception. Затем мы заменяем голову новым набором полносвязных слоев со случайными инициализациями. Оттуда все слои ниже головы замораживаются, поэтому их веса не могут быть обновлены (т. е. обратный проход в обратном распространении не достигает их).

Затем мы обучаем сеть, используя очень небольшую скорость обучения, чтобы новый набор слоев FC мог начать изучать шаблоны из ранее изученных слоев CONV ранее в сети. При желании мы можем разморозить остальную часть сети и продолжить обучение. Применение тонкой настройки позволяет нам применять предварительно обученные сети для распознавания классов, на которых они изначально не обучались; кроме того, этот метод может обеспечить более высокую точность, чем извлечение признаков.

В оставшейся части этой главы мы более подробно обсудим тонкую настройку, в том числе сетевую хирургию. Мы закончим тем, что приведем пример применения тонкой настройки к набору данных Flowers-17 и превзойдем все другие подходы, которые мы пробовали в этой книге до сих пор.

5.1 Перенос обучения и точная настройка

Тонкая настройка — это тип трансферного обучения. Мы применяем тонкую настройку к моделям глубокого обучения, которые уже были обучены на заданном наборе данных. Как правило, эти сети представляют собой современные архитектуры, такие как VGG, ResNet и Inception, которые были обучены на наборе данных ImageNet.

Как мы узнали из главы 3, посвященной извлечению признаков, эти сети содержат богатые дискриминационные фильтры, которые можно использовать для наборов данных и меток классов, помимо тех, на которых они уже обучены. Однако вместо того, чтобы просто применять извлечение признаков, мы собираемся провести сетевую хирургию и изменить фактическую архитектуру, чтобы мы могли переобучить части сети.

Если это звучит как что-то из плохого фильма ужасов; не волнуйтесь, крови не будет – но мы повеселимся и многому научимся в ходе наших экспериментов. Чтобы понять, как работает тонкая настройка, рассмотрите рисунок 5.1 (слева), на котором показаны слои сети VGG16. Как мы знаем, последний набор слоев (то есть «голова») — это наши полностью связанные слои вместе с нашим классификатором softmax. При выполнении тонкой настройки мы фактически удаляем голову из сети, как и при извлечении признаков (посередине). Однако, в отличие от извлечения признаков, когда мы выполняем точную настройку, мы фактически создаем новую полностью подключенную головку и размещаем ее поверх исходной архитектуры (справа).

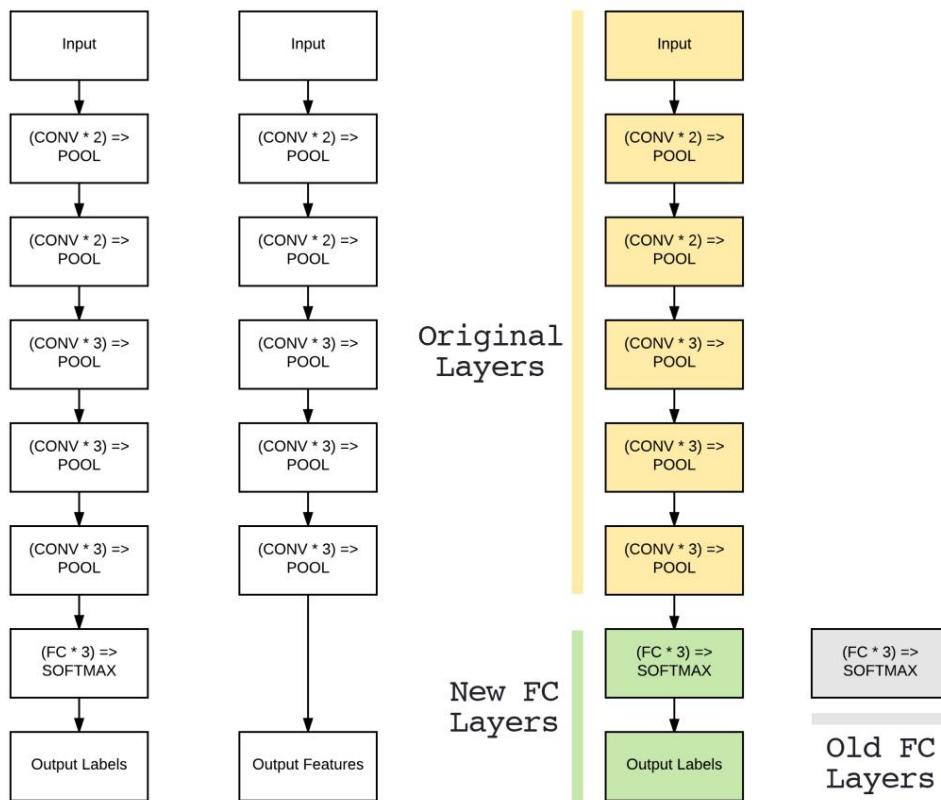


Рисунок 5.1: Слева: исходная сетевая архитектура VGG16. Посередине : удаление слоев FC из VGG16 и обработка последнего слоя POOL в качестве экстрактора признаков. Справа: удаление исходных слоев FC и замена их новой головкой FC . Затем эти новые слои FC можно точно настроить для конкретного набора данных (старые слои FC больше не используются).

В большинстве случаев ваша новая головка FC будет иметь меньше параметров, чем исходная; однако это действительно зависит от вашего конкретного набора данных. Новая головка FC инициализируется случайным образом (как и любой другой слой в новой сети) и подключается к телу исходной сети, и мы готовы к обучению.

Однако есть проблема — наши слои CONV уже изучили богатые, различающие фильтры , в то время как наши слои FC являются совершенно новыми и полностью случайными. Если мы позволим градиенту распространяться обратно от этих случайных значений по всему телу нашей сети, мы рискуем разрушить эти

мощные функции. Чтобы обойти это, вместо этого мы даем нашей головке FC «разогреться», (по иронии судьбы) «замораживая» все слои в теле сети (я говорил вам, что здесь хорошо работает аналогия с трупом), как на рис. 5.2 (слева).

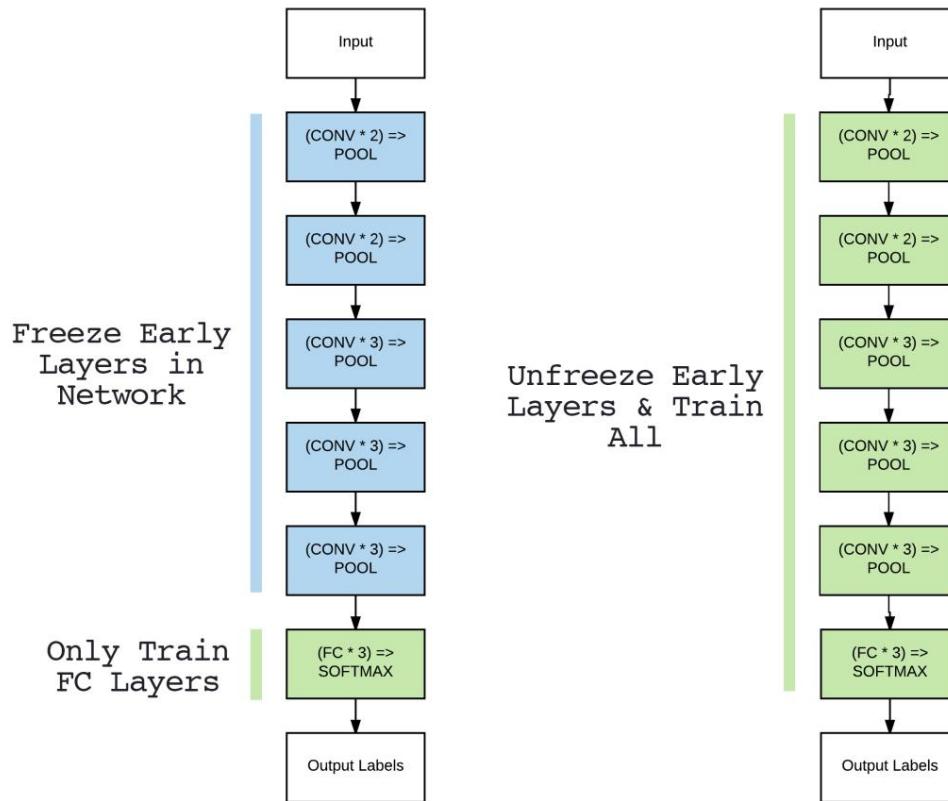


Рисунок 5.2: Слева: когда мы начинаем процесс тонкой настройки, мы замораживаем все слои CONV в сети и разрешаем обратное распространение градиента только через слои FC . Это позволяет нашей сети «разогреться». Справа: после того , как слои FC получили возможность прогреться, мы можем разморозить все слои в сети и также позволить тонкой настройке каждого из них.

Обучающие данные распространяются по сети, как обычно; однако обратное распространение останавливается после слоев FC , что позволяет этим слоям начать изучать шаблоны из сильно различающихся слоев CONV . В некоторых случаях мы никогда не сможем разморозить тело сети, так как наша новая головка FC может получить достаточную точность. Однако для некоторых наборов данных часто выгодно разрешить изменение исходных слоев CONV в процессе тонкой настройки (рис. 5.2, справа).

После того, как головка FC начала изучать шаблоны в нашем наборе данных, приостановить обучение, разморозить тело, а затем продолжить обучение, но с очень небольшой скоростью обучения — мы не хотим резко отклоняться от наших фильтров CONV . Затем обучение продолжают до тех пор, пока не будет достигнута достаточная точность.

Тонкая настройка — это супермощный метод получения классификаторов изображений из предварительно обученных CNN на пользовательских наборах данных, в большинстве случаев даже более мощный, чем извлечение признаков. Недостатком является то, что точная настройка может потребовать немного больше работы, и ваш выбор параметров головки FC действительно играет большую роль в точности сети — здесь вы не можете строго полагаться на методы регуляризации, так как ваша сеть уже была предварительно обучена, и вы не можете отклоняться от регуляризации, уже осуществляющейся ею.

Во-вторых, для небольших наборов данных может быть сложно заставить вашу сеть начать «обучение» с

«холодный» запуск FC , поэтому сначала мы замораживаем тело сети. Тем не менее, прохождение этапа разогрева может оказаться сложной задачей и может потребовать от вас использования оптимизаторов, отличных от SGD (описано в главе 7). В то время как тонкая настройка требует немного больше усилий, если она сделана правильно, вы почти всегда будете наслаждаться более высокой точностью.

5.1.1 Индексы и слои Перед

выполнением сетевых операций нам необходимо знать имя слоя и индекс каждого слоя в данной модели глубокого обучения. Нам нужна эта информация, поскольку нам потребуется «заморозить» и «разморозить» определенные слои в предварительно обученной CNN. Не зная заранее имен и индексов слоев , мы бы «резали вслепую», выйдя из-под контроля хирургом без плана игры. Если вместо этого мы потратим несколько минут на изучение сетевой архитектуры и реализации, мы сможем лучше подготовиться к операции.

Давайте продолжим и посмотрим на имена и индексы слоев в VGG16. Откройте новый файл, назовите его inspect_model.py и вставьте следующий код:

```

1 # импортируем необходимые пакеты 2
из keras.applications import VGG16 3 import
argparse
4
5 # построить разбор аргумента и разобрать аргументы 6 ap =
argparse.ArgumentParser() 7 ap.add_argument("-i", "--include-top", type=int,
default=1, help="независимо от того, включить верхнюю часть CNN") 9 args =
8     vars(ap.parse_args())
10
11 # загружаем сеть VGG16
12 print("[INFO] загружает сеть...") 13 model =
VGG16(weights="imagenet",
14     include_top=args["include_top"] > 0) 15
print("[INFO] показывает слои...")
16
17 # перебираем слои в сети и выводим их на 18 # консоль
19 для (i, слой) в enumerate(model.layers): print("[INFO] {})
20     \t{}".format(i, layer.__class__.__name__)

```

Строки 2 импортируют нашу реализацию VGG16 Keras , сеть, которую мы будем изучать и готовить к операции. Строки 6-9 анализируют наши аргументы командной строки. Здесь необходим единственный переключатель --include-top , который используется для указания, следует ли включать главу сети в сводку модели.

Строки 23 и 24 загружают VGG16 предварительно обученными весами ImageNet с диска — головка сеть опционально включена. Наконец, строки 19 и 20 позволяют нам исследовать нашу модель.

Для каждого слоя в сети мы печатаем соответствующий индекс, т.е. Учитывая эту информацию, мы будем знать индекс того, где начинается головка FC (и где ее заменить нашей новой головкой FC).

Чтобы исследовать архитектуру VGG16, просто выполните следующую команду:

```

$ python inspect_model.py [INFO]
показывает слои...
[INFO] 0 InputLayer [INFO] 1
Conv2D [INFO] 2
    Conv2D

```

[ИНФОРМАЦИЯ] 3	MaxPooling2D
[ИНФОРМАЦИЯ] 4	Conv2D
[ИНФОРМАЦИЯ] 5	Conv2D
[ИНФОРМАЦИЯ] 6	MaxPooling2D
[ИНФОРМАЦИЯ] 7	Conv2D
[ИНФОРМАЦИЯ] 8	Conv2D
[ИНФОРМАЦИЯ] 9	Conv2D
[ИНФОРМАЦИЯ] 10	MaxPooling2D
[ИНФО] 11	Conv2D
[ИНФО] 12	Conv2D
[ИНФОРМАЦИЯ] 13	Conv2D
[ИНФО] 14	MaxPooling2D
[ИНФОРМАЦИЯ] 15	Conv2D
[ИНФОРМАЦИЯ] 16	Conv2D
[ИНФО] 17	Conv2D
[ИНФО] 18	MaxPooling2D
[ИНФО] 19	Сгладить
[ИНФОРМАЦИЯ] 20	Плотный
[ИНФО] 21	Плотный
[ИНФОРМАЦИЯ] 22	Плотный

Здесь мы видим, что слои 20-22 являются нашими полносвязными слоями. Для проверки повторно запустим `inspect_model.py`, но этот раз указав переключатель `--include-top -1`, который оставит с головы ФК:

```
$ python inspect_model.py --include-top -1
[INFO] показаны слои...
[ИНФОРМАЦИЯ] 0 InputLayer
[ИНФО] 1 Conv2D
[ИНФОРМАЦИЯ] Conv2D
2 [ИНФОРМАЦИЯ] MaxPooling2D
3 [ИНФОРМАЦИЯ] Conv2D
4 [ИНФОРМАЦИЯ] Conv2D
5 [ИНФОРМАЦИЯ] MaxPooling2D
6 [ИНФОРМАЦИЯ] Conv2D
7 [ИНФОРМАЦИЯ] Conv2D
8 [ИНФОРМАЦИЯ] Conv2D
9 [ИНФОРМАЦИЯ] MaxPooling2D
10 [ИНФОРМАЦИЯ] Conv2D
11 [ИНФОРМАЦИЯ] Conv2D
12 [ИНФОРМАЦИЯ] Conv2D
13 [ИНФОРМАЦИЯ] MaxPooling2D
14 [ИНФОРМАЦИЯ] Conv2D
15 [ИНФОРМАЦИЯ] Conv2D
16 [ИНФОРМАЦИЯ] Conv2D
17 [ИНФОРМАЦИЯ] 18 MaxPooling2D
```

Обратите внимание, что последним слоем в сети теперь является слой POOL (точно так же, как в главе 3, посвященной функциям экстракция). Эта часть сети послужит отправной точкой для тонкой настройки.

5.1.2 Сетевая хирургия

Прежде чем мы сможем заменить голову предварительно обученной CNN, нам нужно что-то, чем можно ее заменить — следовательно, нам нужно определить нашу собственную полносвязную главу сети. Для начала создайте новый файл с именем `fcheadnet.py` в подмодуле `nn.conv pyimagesearch`:

```

--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ио
| |--- нн
| | |--- __init__.py
| | |--- конв.
| | | |--- __init__.py
| | | |--- lenet.py
| | | |--- minivggnet.py
| | | |--- fheadnet.py
| | | |--- мелкая сеть.py
| |--- предварительная обработка
| |--- утилиты

```

Затем откройте fheadnet.py и вставьте следующий код:

```

1 # импортируем необходимые пакеты
2 из keras.layers.core import Dropout
3 из keras.layers.core import Flatten
4 из keras.layers.core импорт Плотный

```

Строки 2-4 импортируют необходимые пакеты Python. Как видите, эти три пакета обычно используется только для полно связных сетей (исключением является слой отсева).

Далее давайте определим класс FCEadNet:

```

6 класс FCHeadNet:
7     @статический метод
8     def build (baseModel, классы, D):
9         # инициализируем модель головы, которая будет размещена поверх
10        # база, затем добавляем слой FC
11        headModel = baseModel.output
12        headModel = Flatten(name="flatten")(headModel)
13        headModel = Плотная (D, активация = "relu") (headModel)
14        headModel = Dropout (0,5) (headModel)
15
16        # добавить слой softmax
17        headModel = Dense (классы, активация = "softmax") (headModel)
18
19        # вернуть модель
20        вернуть головную модель

```

Как и в наших предыдущих реализациях сети, мы определяем метод сборки , ответственный за для построения фактической сетевой архитектуры. Этот метод требует три параметра: BaseModel (тело сети), общее количество классов в нашем наборе данных и, наконец , D , количество узлов в полно связном слое.

Строка 11 инициализирует headModel , которая отвечает за соединение нашей сети с остальное тело, BaseModel.output. Оттуда строки 12-17 строят очень простой полно связный архитектура:

ВХОД => FC => RELU => DO => FC => SOFTMAX

Опять же, эта полностью подключеная головка очень упрощена по сравнению с оригинальной головкой от VGG16 , которая состоит из двух наборов 4096 слоев FC . Тем не менее, для большинства задач тонкой настройки вы не пытаетесь воспроизвести исходный заголовок сети, а скорее упрощаете его, чтобы его было легче настраивать — чем меньше параметров в голове, тем больше вероятность того, что мы будем правильно настроить сеть на новую задачу классификации. Наконец, строка 20 возвращает вновь созданную головку FC вызывающей функции.

Как мы увидим в следующем разделе, мы заменим головку VGG16 нашим вновь определенным FCEadNet через сетевую хирургию.

5.1.3 Тонкая настройка от начала до конца

Пришло время применить тонкую настройку от начала до конца. Откройте новый файл, назовите его `finetune_flowers17.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 _ _ _
_ _ _ _ _ pyimagesearch.nn.conv импортировать FCHeadNet
9 из keras.preprocessing.image импортировать ImageDataGenerator 10 из
keras.optimizers импортировать RMSprop 11 из keras.optimizers импортировать SGD
12 из keras.applications импортировать VGG16 13 из keras.layers импортировать
Input 14 из keras.models import Модель 15 из путей импорта imutils 16 import numpy
as np 17 import argparse 18 import os
```

Строки 2-18 требуют импорта наших пакетов Python, большего количества пакетов, которые мы видели ранее в наших предыдущих примерах (хотя многие из них нам уже знакомы). Строки 5-7 импортируют наши препроцессоры изображений вместе с загрузкой нашего набора данных. Стока 8 импортирует нашу недавно определенную FCHeadNet для замены головы VGG16 (строка 12). Импорт класса ImageDataGenerator в строке 9 означает , что мы будем применять увеличение данных к нашему набору данных.

Строки 10 и 11 импортируют наши оптимизаторы, необходимые нашей сети для фактического изучения шаблонов из входных данных. Мы уже довольно хорошо знакомы с SGD, но еще не рассмотрели RMSprop — мы отложим обсуждение продвинутых методов оптимизации до главы 7, а пока просто поймем, что RMSprop часто используется в ситуациях, когда нам нужно быстро получить приемлемую производительность (как в случае, когда мы пытаемся «разогреть» набор слоев FC).

Строки 13 и 14 импортируют два класса, необходимых при тонкой настройке с помощью Keras — Input и Модель. Нам потребуются оба из них при выполнении сетевой хирургии.

Давайте продолжим и проанализируем наши аргументы командной строки:

```
20 # построить разбор аргумента и разобрать аргументы 21 ap =
argparse.ArgumentParser() 22 ap.add_argument("-d", "--dataset",
required=True,
23         help="путь к входному набору
данных") 24 ap.add_argument("-m", "--model", required=True,
```

```
25     help="путь к выходной модели")
26 args = vars(ap.parse_args())
```

Нам потребуются два аргумента командной строки для нашего сценария: --dataset, путь к входному каталогу, содержащему набор данных Flowers-17, и --model, путь к нашим выходным сериализованным весам после обучения.

Мы также можем инициализировать ImageDataGenerator, отвечающий за увеличение данных при обучении нашей сети:

```
28 # создать генератор изображений для увеличения данных 29 aug
= ImageDataGenerator(rotation_range=30, width_shift_range=0.1,
30     height_shift_range = 0.1, shear_range = 0.2, zoom_range = 0.2,
31     horizontal_flip = True, fill_mode = «ближайший»)
```

Как я упоминал в главе 2, почти во всех случаях следует применять аугментацию данных, поскольку она редко снижает точность и часто помогает увеличить ее и избежать переобучения. То же самое особенно верно для тонкой настройки, когда у нас может не хватить данных для обучения глубокой CNN с нуля.

Следующий блок кода обрабатывает получение imagePaths с диска вместе с анализом classNames из путей к файлам:

```
33 # получить список изображений, которые мы будем описывать, затем извлечь
34 # имена меток классов из путей к изображениям 35 print("[INFO] loading
images...") 36 imagePaths = list(paths.list_images(args ["набор данных"])) 37 classNames
= [pt.split(os.path.sep)[-2] для pt в imagePaths] 38 classNames = [str(x) для x в
np.unique(classNames)]
```

Опять же, мы делаем предположение, что наш входной набор данных имеет следующую структуру каталогов:

имя_набора_данных/{имя_класса}/example.jpg

Следовательно, мы можем использовать разделитель пути, чтобы легко (и удобно) извлечь метку класса из пути к файлу.

Теперь мы готовы загрузить наш набор данных изображений с диска:

```
40 # инициализируем препроцессоры
изображений 41 aap = AspectAwarePreprocessor(224,
224) 42 iap = ImageToArrayPreprocessor()
43
44 # загрузить набор данных с диска, затем масштабировать необработанные
пиксельные интенсивности до 45 # диапазон [0, 1] 46 sdl =
SimpleDatasetLoader(preprocessors=[aap, iap]) 47 (data, labels) = sdl.load(imagePaths,
verbose =500) 48 данных = data.astype("с плавающей запятой") / 255.0
```

Строки 41 и 42 инициализируют наши препроцессоры изображений. Мы изменим размер всех входных изображений до 224× 224 пикселей (с сохранением исходного соотношения сторон изображения), необходимого входного размера для сети VGG16. Строки 46 и 47 затем применяют наши препроцессоры изображений для загрузки данных и меток с диска.

Далее создадим наши обучающие и тестовые сплиты (75% данных для обучения, 25% для тестирования) и однократное кодирование меток:

```

50 # разбить данные на обучающие и тестовые сплиты, используя 75% от 51 # данные
для обучения и оставшиеся 25% для тестирования 52 (trainX, testX, trainY, testY) =
train_test_split(data, labels, test_size=0.25, random_state= 42)

53
54
55 # преобразовать метки из целых чисел в векторы 56 trainY
= LabelBinarizer().fit_transform(trainY) 57 testY =
LabelBinarizer().fit_transform(testY)

```

А вот и самое интересное — выполнение сетевой операции:

```

59 # загрузите сеть VGG16, убедившись, что наборы головных слоев FC отключены 60 #
61 modelBuilder = VGG16(weights="imagenet", include_top=False,
62           input_tensor = Ввод (форма = (224, 224, 3)))
63
64 # инициализируем новый заголовок сети, набор слоев FC 65 # за которым
следует классификатор softmax 66 headModel = FCHeadNet.build(baseModel,
len(classNames), 256)
67
68 # поместите головную модель FC поверх базовой модели -- 69 # она станет
фактической моделью, которую мы будем обучать.
70 модель = модель (входы = modelBuilder.input, выходы = headModel)

```

Строки 61 и 62 загружают архитектуру VGG16 с диска, используя предоставленные предварительно обученные веса ImageNet. Мы намеренно опустим голову VGG16, так как заменим ее нашей собственной FCEadNet. Мы также хотим явно определить input_tensor равным 224× 224× 3 пикселя (опять же, предполагая порядок каналов), иначе мы столкнемся с ошибками при попытке обучить нашу сеть, поскольку формы объемов не будут совпадать.

Строка 66 создает экземпляр FCHeadNet , используя тело modelBuilder в качестве входных данных, len(classNames) в качестве общего количества меток классов (17 в случае Flowers-17), а также 256 узлов на уровне FC.

Фактическая «операция» выполняется в строке 70 , где мы создаем новую модель, используя тело VGG16 (baseModel.input) в качестве входных данных и headModel в качестве выходных данных. Однако мы еще не готовы обучать нашу сеть — имейте в виду, что ранее в этой главе я упоминал, что нам нужно заморозить веса в теле, чтобы они не обновлялись на этапе обратного распространения.

Мы можем выполнить это замораживание, установив для параметра .trainable значение False для каждого слоя в modelBuilder:

```

72 # перебрать все слои в базовой модели и заморозить их, чтобы они 73 # *не*
обновлялись в процессе обучения 74 для слоя в modelBuilder.layers: layer.trainable = False

```

75

Теперь, когда мы соединили голову с телом и заморозили слои в теле, мы можем согреть до нового главы сети:

```

77 # скомпилируем нашу модель (это нужно сделать после того, как мы настроим
наши 78 # слои как необучаемые 79 print("[INFO] компиляция модели..."))

```

```

80 opt = RMSprop(lr=0.001) 81
model.compile(loss="categorical_crossentropy", оптимизатор=opt,
82     метрики=["точность"])
83
84 # обучаем головку сети в течение нескольких эпох (все остальные 85 # слои заморожены) --
# это позволит новым слоям 86 # FC инициализироваться с фактическими "выученными"
значениями.
87 # по сравнению с чисто
88 print("[INFO] training head...") 89
model.fit_generator(aug.flow(trainX, trainY, batch_size=32),
90     validation_data=(testX, testY), epochs=25,
91     steps_per_epoch=len(trainX) // 32, verbose=1)

```

Строка 80 инициализирует оптимизатор RMSprop — алгоритм, который мы подробнее обсудим в главе 7. Обратите внимание, как мы используем небольшую скорость обучения $1e - 3$, чтобы разогреть головку FC . При тонкой настройке вы почти всегда будете использовать скорость обучения, которая на один, если не на несколько порядков меньше, чем исходная скорость обучения, используемая для обучения сети.

Строки 88-91 затем обучают нашу новую головку FC , используя наш метод увеличения данных. Опять же, имейте в виду, что в то время как каждое изображение полностью распространяется вперед, градиенты распространяются только частично обратно — обратное распространение заканчивается после слоев FC , поскольку наша цель здесь — только «разогреть» голову, а не изменить веса. в теле сети. Здесь мы позволяем фазе разминки тренироваться в течение 25 эпох. Обычно вы позволяете своей головке FC прогреваться в течение 10-30 эпох, в зависимости от вашего набора данных.

После фазы прогрева мы сделаем паузу, чтобы оценить производительность сети на тестовом наборе:

```

93 # оценить сеть после инициализации
94 print("[ИНФО] оценка после инициализации...") 95 прогнозы = model.predict(testX,
batch_size=32) 96 print(classification_report(testY.argmax(axis=1),
97     прогнозы.argmax(ось = 1), target_names = classNames))

```

Приведенный выше код позволит нам сравнить эффекты тонкой настройки до и после прогрева головы.

Теперь, когда наши слои FC частично обучены и инициализированы, давайте разморозим некоторые из CONV . слоев в теле и сделать их тренируемыми:

```

99 # теперь, когда слои головного FC обучены/инициализированы, давайте 100
# разморозим окончательный набор слоев CONV и сделаем их обучаемыми 101
для слоя в modelBuilder.layers[15:]: layer.trainable = True
102

```

Сделать данный слой в теле снова обучаемым — это пример установки параметра .trainable в значение True для данного слоя. В некоторых случаях вы захотите позволить тренироваться всему телу ; однако для более глубоких архитектур со многими параметрами, такими как VGG, я предлагаю только разморозить верхние слои CONV , а затем продолжить обучение. Если точность классификации продолжает улучшаться (без переобучения), вы можете рассмотреть возможность размораживания большего количества слоев в теле.

К этому моменту у нас должно быть теплое начало тренировки, поэтому мы переключимся на SGD (опять же с небольшая скорость обучения) и продолжить обучение:

```

104 # чтобы изменения в модели вступили в силу, нам нужно перекомпилировать 105 #
модель, на этот раз с использованием SGD с *очень* небольшой скоростью обучения
106 print("[INFO] re-compiling model...") 107 opt = SGD(lr=0.001) 108
model.compile(loss="categorical_crossentropy", оптимизатор=opt,
109     метрики=["точность"])
110
111 # снова обучить модель, на этот раз с тонкой настройкой *обоих* финального набора
112 # слоев CONV вместе с нашим набором слоев FC 113 print("[INFO] Fine Tuning model...")
114 model.fit_generator (август.поток(trainX, trainY, batch_size=32),
115     validation_data=(testX, testY), epochs=100,
116     steps_per_epoch=len(trainX) // 32, verbose=1)

```

На этот раз мы разрешаем нашей сети обучать более 100 эпох, позволяя фильтрам CONV адаптироваться к лежащие в основе закономерности в обучающих данных.

Наконец, мы можем оценить нашу точно настроенную сеть, а также сериализовать веса на диск:

```

118 # оценить сеть на доработанной модели
119 print("[ИНФО] оценка после тонкой настройки...") 120
прогнозы = model.predict(testX, batch_size=32) 121
print(classification_report(testY.argmax(axis=1),
122     прогнозы.argmax(ось = 1, target_names = classNames))
123
124 # сохранить модель на диск
125 print("[INFO] сериализуемая модель...") 126
model.save(args["model"])

```

Чтобы выполнить сетевую операцию и настроить VGG16 на наборе данных Flowers-17, просто выполните следующую команду:

```
$ python finetune_flowers17.py --dataset ../datasets/flowers17/images \ --model flowers17.model
[INFO] загрузка изображений...
[INFO] обработано 500/1360
[INFO] обработано 1000/1360
[INFO] компиляция модели...
[INFO] Тренировочная голова...
Эпоха 1/25
10 с - потеря: 4,8957 - акк: 0,1510 - val_loss: 2,1650 - val_acc: 0,3618
...
Эпоха 10/25
10 с - потеря: 1,1318 - акк: 0,6245 - val_loss: 0,5132 - val_acc: 0,8441
...
Эпоха 23/25
10 с - потеря: 0,7203 - акк: 0,7598 - val_loss: 0,4679 - val_acc: 0,8529 Эпоха 24/25 10 с -
потеря: 0,7355 - акк: 0,7520 - val_loss: 0,4268 - val_acc: 0,8853 - потеря: 25/25 с 0.7504 - acc:
0.7598 - val_loss: 0.3981 - val_acc: 0.8971 [INFO] оценка после инициализации...
```

колокольчик	0,75	1,00	0,86	18
лютик мать-	0,94	0,85	0,89	20
и-мачеха	0,94	0,85	0,89	20
первоцвет	0,70	0,78	0,74	18
крокус	1,00	0,80	0,89	20
нарцисс	0,87	0,96	0,91	27
	0,90	0,95	0,93	20
ромашка одуванчик	0,96	0,96	0,96	23
рябчатый ирис	1,00	0,86	0,93	22
	1,00	0,95	0,98	21
lilyvalley	0,93	0,93	0,93	15
	0,83	1,00	0,90	19
анютины	0,88	0,96	0,92	23
глазки	1,00	0,96	0,98	23
подснежник	0,90	1,00	0,95	19
	0,86	0,38	0,52	16
подсолнух тигровая лилия	0,83	0,94	0,88	16
среднее / общее	0,90	0,90	0,89	340

Обратите внимание, как наша начальная точность чрезвычайно низка для первой эпохи (~ 36%) во время теплого периода. фаза вверх. Этот результат связан с тем, что слои FC в нашей новой голове инициализируются случайным образом. и все еще пытаюсь изучить шаблоны из ранее обученных фильтров CONV . Однако точность быстро растет – к 10 эпохе мы выше 80% точности классификации, а к концу 25 эпохи мы достигли почти 90% точности.

Теперь, когда наш FCHeadNet получил теплый старт, мы переключаемся на SGD и размораживаем первый набор слоев CONV в теле, позволяющий сети обучаться еще 100 эпох. Точность продолжает улучшаться, вплоть до 95% точности классификации, выше, чем 93%, которые мы получили используя извлечение признаков:

```

1 ...
2 [INFO] перекомпиляция модели...
3 [INFO] тонкая настройка модели...
4 Эпоха 1/100
5 12 с - убыток: 0,5127 - акк: 0,8147 - знач_убыток: 0,3640 - знач_акк: 0,8912
6 ...
7 Эпоха 99/100
8 12 с - убыток: 0,1746 - акк: 0,9373 - val_loss: 0,2286 - val_acc: 0,9265
9 Эпоха 100/100
10 12 с - убыток: 0,1845 - акк: 0,9402 - знач_убыток: 0,2019 - знач_акк: 0,9412
11 [INFO] оценка после тонкой настройки...
12 поддержка точного отзыва f1-score
13
14    колокольчик      0,94      0,94      0,94      18
15    лютик мать-      0,95      1,00      0,98      20
16    и-мачеха         1,00      0,90      0,95      20
17    первоцвет        0,85      0,94      0,89      18
18    крокус          0,90      0,90      0,90      20
19    нарцисс         1,00      0,78      0,88      27
20    ромашка одуванчик 0,96      1,00      0,98      23
21    рябчатых ириса   1,00      0,95      0,98      22
22    лилии           1,00      0,95      0,98      21
23    лилии           1,00      0,95      0,98      15
24    лилии           1,00      0,93      0,97      15

```

5.2 Резюме

25	анютины	1,00	1,00	1,00	19
26	глазки	0,92	0,96	0,94	23
27	подснежник подсолнух	0,96	1,00	0,98	23
28	тигровая	0,90	1,00	0,95	19
29	лилия	0,70	0,88	0,78	16
30	тюльпан ветрянка	0,94	0,94	0,94	16
31					
32	в среднем / всего	0,95	0,94	0,94	340

Дополнительную точность можно получить, выполняя более агрессивную аугментацию данных и постоянно размежевывая все больше и больше блоков CONV в VGG16. В то время как тонкая настройка, безусловно, более работы, чем извлечение признаков, это также позволяет нам настраивать и изменять веса в нашей CNN до конкретный набор данных - то, что не позволяет извлечь признаков. Таким образом, когда дано достаточно обучающие данные, рассмотрите возможность тонкой настройки, так как вы, вероятно, получите более высокую точность классификации чем простое извлечение признаков.

5.2 Резюме

В этой главе мы обсудили второй тип трансферного обучения — тонкую настройку. Доводочные работы по замена полностью подключенной головки сети новой, случайно инициализированной головкой. Слои в теле исходной сети замораживаются, пока мы обучаем новые слои FC.

Как только наша сеть начнет получать достаточную точность, это означает, что уровни FC запустились. Чтобы изучить шаблоны как из (1) базовых данных обучения, так и (2) из ранее обученного CONV фильтры ранее в сети, мы размежевываем часть (если не все) тела — тогда тренировки разрешаются продолжать.

Применение тонкой настройки — чрезвычайно мощный метод, поскольку нам не нужно обучать всю сеть с нуля. Вместо этого мы можем использовать уже существующие сетевые архитектуры, такие как современные модели, обученные на наборе данных ImageNet, которые состоят из богатого дискриминационного набора фильтров. Используя эти фильтры, мы можем «быстро начать» наше обучение, позволяя нам выполнять сетевые хирургии, что в конечном итоге приводит к более точному переносу модели обучения с меньшими усилиями (и головная боль), чем обучение с нуля.

Для получения более практических примеров трансферного обучения и тонкой настройки обязательно обратитесь к ImageNet Bundle, в котором я демонстрирую, как:

1. Узнавать марку и модель автомобиля.
2. Автоматически определять и исправлять ориентацию изображения.

6. Повышение точности с помощью сетевых ансамблей

В этой главе мы рассмотрим концепцию ансамблевых методов, процесс взятия нескольких классификаторов и объединения их в один большой метаклассификатор. Усредняя несколько моделей машинного обучения вместе, мы можем превзойти (то есть достичь более высокой точности), используя только одну модель, выбранную случайнным образом. На самом деле, почти все современные публикации, которые вы читали, которые участвуют в конкурсе ImageNet, сообщают о своих лучших результатах по ансамблям сверточных нейронных сетей.

Мы начнем эту главу с обсуждения неравенства Дженсена, на котором основаны методы теоретического ансамбля. Оттуда я продемонстрирую, как обучить несколько CNN из одного сценария и оценить их производительность. Затем мы объединим эти CNN в один метаклассификатор и заметим повышение точности.

6.1 Методы ансамбля

Термин «методы ансамбля» обычно относится к обучению «большого» количества моделей (где точное значение «большого» зависит от задачи классификации) и последующему объединению их выходных прогнозов посредством голосования или усреднения для повышения точности классификации. На самом деле ансамблевые методы вряд ли специфичны для глубокого обучения и сверточных нейронных сетей. Мы используем ансамблевые методы в течение многих лет. Такие методы, как AdaBoost [18] и Random Forests [19], являются типичными примерами ансамблевых методов.

В случайных лесах мы обучаем несколько деревьев решений [20, 21] и используем наш лес для прогнозирования. Как видно из рисунка 6.2 (слева), наш случайный лес состоит из нескольких объединенных вместе деревьев решений. Каждое дерево решений «голосует» за то, какой, по его мнению, должна быть окончательная классификация. Эти голоса сводятся в таблицу метаклассификатором, и категория с наибольшим количеством голосов выбирается в качестве окончательной классификации.

Та же концепция может быть применена к глубокому обучению и сверточной нейронной сети. Здесь мы обучаем несколько сетей, а затем просим каждую сеть вернуть вероятности для каждой метки класса с учетом точки входных данных (рис. 6.2, слева). Эти вероятности усредняются вместе, и получается окончательная классификация. Чтобы понять, почему усреднение прогнозов по нескольким моделям работает, нам сначала нужно обсудить неравенство Дженсена. Затем мы предоставим код Python и Keras для реализации ансамбля CNN и сами убедимся, что точность классификации действительно увеличивается.

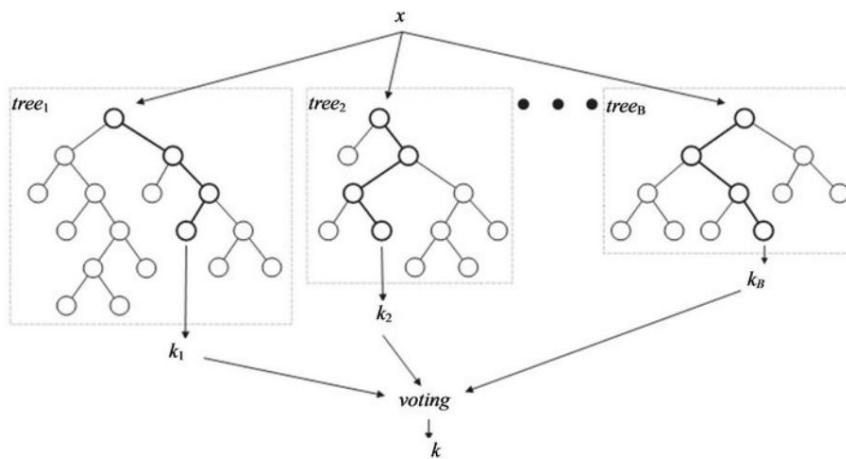


Рисунок 6.1: Случайный лес состоит из нескольких деревьев решений. Результаты каждого дерева решений усредняются вместе для получения окончательной классификации. Изображение воспроизведено из Nguyen et al. [22]

6.1.1 Неравенство Дженсена В самых

общих чертах ансамбль — это конечный набор моделей, которые можно использовать для получения более высокой средней точности предсказания, чем при использовании одной модели в наборе ансамбля. Основополагающая работа Диттериха [23] детализирует теорию того, почему ансамблевые методы обычно могут обеспечивать более высокую точность, чем одна модель.

Работа Диттериха основана на неравенстве Дженсена, известном как «разнообразие» или «разложение неоднозначности» в литературе по машинному обучению. Формальное определение неравенства Дженсена гласит, что выпуклый комбинированный (средний) ансамбль будет иметь ошибку, меньшую или равную средней ошибке отдельных моделей. Может случиться так, что одна отдельная модель имеет меньшую ошибку, чем среднее значение всех моделей, но поскольку нет критерия, по которому мы могли бы «выбрать» эту модель, мы можем быть уверены, что среднее значение всех моделей будет работать не хуже, чем выбор любой модели наугад. Короче говоря, мы можем стать лучше, только усредняя наши прогнозы вместе; нам не нужно бояться ухудшения нашего классификатора.

Тем из нас, кто любит наглядные примеры, возможно, неравенство Дженсена и концепцию усреднения модели лучше всего объяснить, попросив вас посмотреть на эту банку с конфетами и угадать, сколько конфет внутри (рис. 6.2, справа).

Сколько конфет вы угадали? 100? 200? 500? Ваше предположение может быть значительно больше или меньше фактического количества конфет в банке. Это может быть очень близко. Или, если вам очень повезет, вы можете угадать точное количество конфет.

Однако в этой игре есть небольшая хитрость — она основана на неравенстве Дженсена. Если бы вы спросили меня, сколько конфет в банке, я бы подошел к вам и всем остальным, кто приобрел копию Deep Learning for Computer Vision with Python, и спросил каждого из них, сколько, по их мнению, конфет. Затем я брал все эти предположения и усреднял их вместе — и использовал это среднее значение в качестве окончательного прогноза.

Теперь может случиться так, что горстка из вас действительно хорошо угадывает и может превзойти средний уровень; однако у меня нет никакого критерия, чтобы определить, кто из вас действительно хороший гадатель. Поскольку я не могу сказать, кто является лучшим угадателем, я вместо этого возьму среднее значение всех, кого я спрошу, и, таким образом, я гарантированно получу не худший результат (в среднем), чем выбор любого из ваших предположений наугад. Я могу не выигрывать игру в угадайку конфет каждый раз, когда мы играем, но я всегда буду в выигрыше; и это, по сути, неравенство Дженсена.

Разница между случаем угадыванием количества конфет и моделями глубокого обучения заключается в том, что

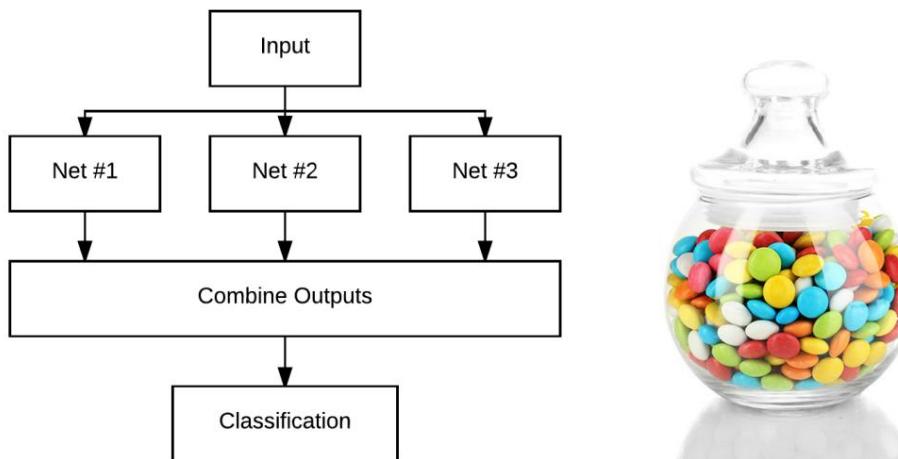


Рисунок 6.2: Слева: ансамбль нейронных сетей состоит из нескольких сетей. При классификации входного изображения точка данных передается в каждую сеть, где она классифицирует изображение независимо от всех других сетей. Затем классификации по сетям усредняются для получения окончательного прогноза. Справа: ансамблевые методы возможны благодаря неравенству Дженсена. Усредняя предположения о количестве конфет в банке, мы можем лучше приблизиться к истинному количеству конфет.

мы предполагаем, что наши CNN работают хорошо и являются хорошими угадывающими (т. е. не случайными угадываниями). Поэтому, если мы усредним результаты этих предикторов вместе, мы часто увидим повышение точности нашей классификации. Именно благодаря этому усовершенствованию вы видите современные публикации по глубокому обучению, обучающие несколько моделей, а затем сообщающие об их наилучшей точности в этих ансамблях.

6.1.2 Построение ансамбля CNN

Первым шагом в построении ансамбля CNN является обучение каждой отдельной CNN. На данный момент в разделе «Глубокое обучение компьютерному зрению с помощью Python» мы видели много примеров обучения одной CNN, но как нам обучить несколько сетей? В общем, у нас есть два варианта: 1. Несколько раз запустить скрипт, который мы используем для обучения одной сети, меняя путь к выходу.

серIALIZОВАННЫЕ ВЕСА МОДЕЛЕЙ ДОЛЖНЫ БЫТЬ УНИКАЛЬНЫМИ ДЛЯ КАЖДОГО ПРОГНОЗА.

2. Создайте отдельный скрипт Python, который использует цикл for для обучения N сетей и выводит сериализованную модель в конце каждой итерации.

Оба метода вполне приемлемы для обучения простого ансамбля CNN. Поскольку нам вполне удобно запускать одну команду для создания одной выходной CNN, давайте попробуем второй вариант, когда один скрипт отвечает за обучение нескольких сетей. Откройте новый файл, назовите его train_models.py и вставьте следующий код:

```
1 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом
 режиме 2 import matplotlib 3 matplotlib.use("Agg")
```

4

```
5 # импортируем необходимые пакеты 6
из sklearn.preprocessing импортируем LabelBinarizer 7 из
sklearn.metrics -----
```

```
11 из keras.datasets import cifar10 12 import
matplotlib.pyplot as plt 13 import numpy as np 14
import argparse 15 import os
```

Строки 2 и 3 импортируют пакет `matplotlib`, а затем устанавливают серверную часть таким образом, чтобы мы могли сохранять графики на диск. Строки 6-15 затем импортируют оставшиеся пакеты Python. Все эти пакеты мы использовали ранее, но ниже я назову наиболее важные: • Стока 8: мы будем обучать несколько моделей MiniVGGNet, чтобы сформировать наш ансамбль. • Стока 9: мы будем использовать класс `ImageDataGenerator` для применения расширения данных при обучении нашей сети. • Строки 10 и 11: наши модели MiniVGGNet будут обучаться на наборе данных CIFAR-10 с использованием оптимизатора SGD.

Сценарию `train_models.py` потребуются два аргумента командной строки, за которыми следует дополнительный необязательный:

```
17 # построить разбор аргумента и разобрать аргументы 18 ap =
argparse.ArgumentParser() 19 ap.add_argument("-o", "--output", required=True,
20         help="путь к выходному каталогу")
21 ap.add_argument("-m", "--models", required=True, help="путь к
22         каталогу выходных моделей")
23 ap.add_argument("-n", "--num-models", type=int, default=5, help="количество
24         моделей для обучения") 25 args = vars(ap.parse_args())
```

Аргумент `--output` будет служить базовым выходным каталогом, в котором мы будем сохранять отчеты о классификации вместе с графиками потерь/точности для каждой из сетей, которые мы будем обучать. Затем у нас есть переключатель `--models`, который управляет путем к выходному каталогу, где мы будем хранить наши сериализованные сетевые веса.

Наконец, аргумент `--num-models` указывает количество сетей в нашем ансамбле. По умолчанию это значение равно 5 сетям. В то время как традиционные методы ансамбля, такие как случайные леса обычно состоят из > 30 деревьев решений (а во многих случаях > 100), мы обычно видим только 5-10 Сверточные нейронные сети в ансамбле — причина в том, что CNN намного обучение требует больше времени и вычислительных ресурсов.

Наш следующий блок кода обрабатывает загрузку набора данных CIFAR-10 с диска, масштабирование пикселей в интенсивностях до диапазона [0,1] и горячее кодирование наших меток классов, чтобы мы могли применить категориальную кросс-энтропию в качестве нашей функции потерь:

```
27 # загрузить данные обучения и тестирования, затем масштабировать их в
диапазоне 28 # [0, 1] 29 ((trainX, trainY), (testX, testY)) = cifar10.load_data() 30 trainX
= trainX.astype("поплавок") / 255,0 31 testX = testX.astype ("поплавок") / 255,0
```

³²
33 # преобразовать метки из целых чисел в векторы 34 lb =
LabelBinarizer() 35 trainY = lb.fit_transform(trainY) 36 testY =
lb.transform(testY)

```

38 # инициализировать имена меток для набора данных CIFAR-10
39 labelNames = ["самолет", "автомобиль", "птица", "кошка", "олень",
40     «собака», «лягушка», «лошадь», «корабль», «грузовик»]

```

Нам также нужно инициализировать наш `ImageDataGenerator`, чтобы мы могли применить увеличение данных к Тренировочные данные CIFAR-10:

```

42 # построить генератор изображений для увеличения данных
43 августа = ImageDataGenerator (rotation_range = 10, width_shift_range = 0,1,
44     height_shift_range=0,1, horizontal_flip=Истина,
45     fill_mode="ближайший")

```

Здесь мы допустим случайный поворот изображений на 10 градусов, смещение на коэффициент 0,1 и случайным образом переворачивается по горизонтали.

Теперь мы готовы обучить каждую отдельную модель MiniVGGNet в ансамбле:

```

47 # цикл по количеству моделей для обучения
48 для i в np.arange(0, args["num_models"]):
49     # инициализируем оптимизатор и модель
50     print("[INFO] модель обучения {} / {}".format(i + 1,
51         аргументы["число_моделей"]))
52     opt = SGD(lr=0,01, затухание=0,01 / 40, импульс=0,9,
53             нестеров = Верно)
54     модель = MiniVGGNet.build (ширина = 32, высота = 32, глубина = 3,
55             классы=10)
56     модель.compile(потеря="categorical_crossentropy", оптимизатор=opt,
57             метрики=["точность"])

```

В строке 48 мы начинаем перебирать количество --пум-моделей для обучения. Стока 52 инициализирует оптимизатор SGD, использующий скорость обучения $\alpha = 0,01$, импульс $\gamma = 0,9$ и стандартный Скорость обучения Keras падает скорость обучения, деленная на общее количество эпох (глава 16, Стартовый набор). Укажем также, что следует использовать ускорения Нестерова. Строки 54-57 затем создайте экземпляр отдельной модели MiniVGGNet и скомпилируйте ее.

Далее давайте обучим сеть и сериализуем ее на диск:

```

59     # обучить сеть
60     H = модель.fit_generator(aug.flow(trainX, trainY, batch_size=64),
61         validation_data=(testX, testY), эпохи=40,
62         steps_per_epoch=len(trainX) // 64, verbose=1)
63
64     # сохранить модель на диск
65     p = [args["модели"], "model_{}.model".format(i)]
66     модель.save (os.path.sep.join (p))

```

Строки 60-62 обучают нашу модель MiniVGGNet, используя метод `fit_generator`. Мы используем `fit_generator` потому что нам нужен метод `.flow` `ImageDataGenerator` для применения увеличения данных. сеть будет обучена в общей сложности 64 эпохи, используя размер пакета 64 . Параметр управляет количеством пакетов в эпоху, то есть просто количеством обучающих выборок. разделить на размер нашей партии.

После того, как сеть закончит обучение, мы строим для нее уникальный выходной путь и сохраняем веса на диск (строки 65 и 66). Давайте также сохраним классификационный_отчет на диск для каждой сети . также, чтобы мы могли проверить производительность после завершения выполнения скрипта:

```

68     # оценить сеть
69     прогнозы = model.predict (testX, batch_size = 64)
70     отчет = отчет_классификации (testY.argmax (ось = 1),
71         прогнозы.argmax(ось=1), target_names=labelNames)
72
73     # сохранить отчет о классификации в файл
74     p = [аргументы ["вывод"], "model_{}.txt".format(i)]
75     f = открыть (os.path.sep.join (p), «w»)
76     f.написать (отчет)
77     f.закрыть()

```

То же самое касается графика наших потерь и точности с течением времени:

```

79     # построить график потери и точности обучения
80     p = [аргументы ["вывод"], "model_{}.png".format(i)]
81     plt.style.use("ggplot")
82     plt.figure()
83     plt.plot(np.arange(0, 40), H.history["потеря"],
84             метка = "поезд_потеря")
85     plt.plot(np.arange(0, 40), H.history["val_loss"],
86             метка = "val_loss")
87     plt.plot(np.arange(0, 40), H.history["acc"],
88             метка = "train_acc")
89     plt.plot(np.arange(0, 40), H.history["val_acc"],
90             метка = "val_acc")
91     plt.title(" Потери при обучении и точность для модели {}".format(i))
92     plt.xlabel("Эпоха #")
93     plt.ylabel("Потери/точность")
94     plt.legend()
95     plt.savefig(os.path.sep.join(p))
96     plt.close()

```

Важно отметить, что мы никогда не приступим сразу к обучению ансамбля — мы сначала проведите серию экспериментов, чтобы определить, какое сочетание архитектуры, оптимизатора и гиперпараметры обеспечивают высочайшую точность для заданного набора данных.

Как только вы достигнете этого оптимального набора комбинаций, мы перейдем к тренировке. несколько моделей для создания ансамбля. Обучение ансамбля в качестве самого первого эксперимента считается преждевременной оптимизацией , поскольку вы не знаете, какая комбинация архитектуры, оптимизатора и гиперпараметры будут работать лучше всего для вашего набора данных.

С учетом сказанного, из главы 15 Starter Bundle мы знаем, что MiniVGGNet обучался с SGD дает разумную точность классификации 83% — мы надеемся, что применяя ансамблевые методы увеличить эту точность.

Чтобы обучить наш набор моделей MiniVGGNet, просто выполните следующую команду:

```
$ python train_models.py --output output --models models
[INFO] тренировочная модель 1/5
[INFO] тренировочная модель 2/5
```

```
[INFO] тренировочная модель 3/5
[INFO] тренировочная модель 4/5
[INFO] тренировочная модель 5/5
```

Поскольку сейчас мы обучаем пять сетей, а не одну, этот скрипт займет в 5 раз больше времени. пробег. Как только он выполнится, взгляните на свой выходной каталог:

```
$ ls вывод/
модель_0.png модель_1.png модель_2.png модель_3.png модель_4.png
модель_0.txt модель_1.txt модель_2.txt модель_3.txt модель_4.txt
```

Здесь вы увидите выходные отчеты о классификации и кривые обучения для каждой из сетей. Используя grep, мы можем легко получить точность классификации каждой сети:

```
$ grep 'среднее / общее' вывод/*.txt
output/model_0.txt:avg/общий вывод/ 0,83 0,83 0,83 10000
model_1.txt:avg/общий вывод/model_2.txt:avg/ 0,83 0,83 0,83 10000
общий вывод/model_3.txt:avg/общий вывод/ 0,83 0,83 0,83 10000
model_4.txt:avg/общий 0,82 0,82 0,82 10000
 0,83 0,83 0,83 10000
```

Четыре из пяти сетей обеспечивают точность классификации 83%, в то время как оставшаяся сеть достигает только 82% точности. Кроме того, глядя на все пять обучающих графиков (рис. 6.3), мы можем увидеть, что каждый набор кривых обучения выглядит примерно одинаково, хотя каждый из них также выглядит уникальным, демонстрируя, что каждая модель MiniVGGNet «обучалась» по-разному.

Теперь, когда мы обучили наши пять индивидуальных ансамблей, пришло время объединить их прогнозы и посмотрите, увеличится ли точность нашей классификации.

6.1.3 Оценка ансамбля

Чтобы построить и оценить наш ансамбль CNN, создайте отдельный файл с именем test_ensemble.py. и вставьте следующий код:

```
1 # импортируем необходимые пакеты
2 из sklearn.preprocessing импорта LabelBinarizer
3 из sklearn.metrics импорта classification_report -
4 из keras.models импортировать load_model
5 из keras.datasets импортировать cifar10
6 импортировать numpy как np
7 импортировать синтаксический анализ
8 импортировать глобус
9 импорт ОС
10
11 # построить аргумент parse и разобрать аргументы
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-m", "--models", required=True,
14     help="путь к каталогу моделей")
15 аргументов = вары (ap.parse_args())
```

Строки 2-9 импортируют наши необходимые пакеты Python, а строки 12-15 анализируют нашу командную строку. аргументы. Нам нужен только один ключ, здесь --models путь к сериализованному веса сети хранятся на диске.

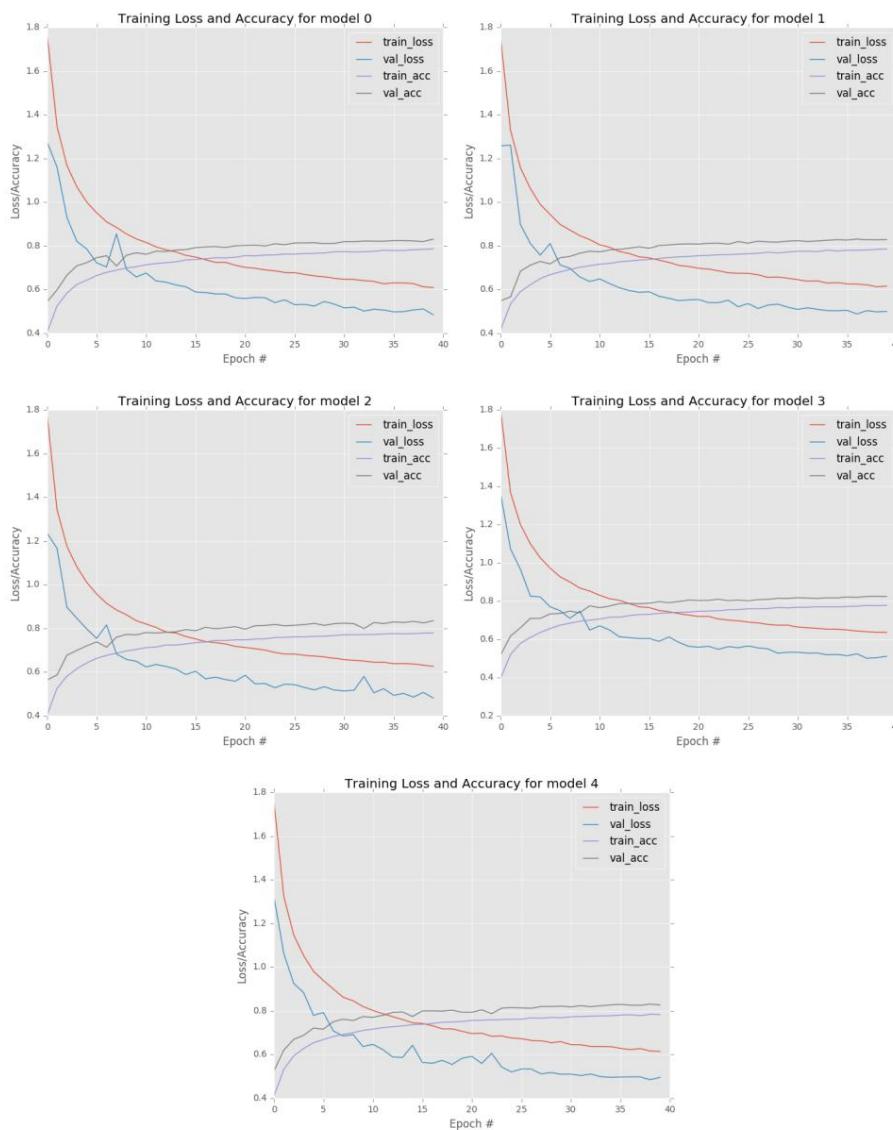


Рисунок 6.3: Графики обучения и проверки для каждой из пяти сетей в нашем ансамбле.

Оттуда мы можем загрузить набор данных CIFAR-10, сохранив только тестовый набор, поскольку мы только оценка (а не обучение) наших сетей:

```
17 # загрузить тестовые данные, затем масштабировать их в диапазоне [0, 1]
18 (testX, testY) = cifar10.load_data()[1] 19 testX = testX.astype("float") / 255.0
```

```
20
21 # инициализировать имена меток для набора данных CIFAR-10
```

```
22 labelNames = ["самолет", "автомобиль", "птица", "кошка", "олень", "собака",
23     "лягушка", "лошадь", "корабль", "грузовик"]
```

```
24
25 # преобразовать метки из целых чисел в векторы 26 lb =
26 LabelBinarizer() 27 testY = lb.fit_transform(testY)
```

Теперь нам нужно собрать пути к нашим предварительно обученным сетям MiniVGGNet, что несложно. достаточно использовать модуль glob, встроенный в Python:

```
29 # создать путь, используемый для сбора моделей, затем инициализировать 30 # список
  моделей 31 modelPaths = os.path.sep.join([args["models"], "*.model"]) 32 modelPaths =
list(glob.glob(modelPaths)) 33 модели = []
```

В строке 31 создается путь с подстановочными знаками (обратите внимание на звездочку «*» в пути к файлу) ко всем файлам .model в каталоге --models . Используя glob.glob в строке 32 , мы можем автоматически найти все пути к файлам внутри --models , которые заканчиваются расширением файла .model . После выполнения строки 32 наш список modelPaths теперь содержит следующие записи:

```
['модели/модель_0.model', 'модели/модель_1.model', 'модели/модель_2.model',
 'модели/модель_3.model', 'модели/модель_4.model']
```

Затем в строке 33 инициализируется список моделей , в которых будут храниться десериализованные сети MiniVGGNet. загружается с диска.

Давайте продолжим и загрузим каждую модель с диска:

```
35 # цикл по пути модели, загрузка модели и добавление ее в 36 # список моделей
```

```
37 для (i, modelPath) в перечислении (modelPaths):
38     print("[INFO] загрузка модели {} / {}".format(i + 1,
39             len(modelPaths)))
40     models.append(load_model(modelPath))
```

В строке 37 мы перебираем каждый из отдельных путей к файлу modelPath . Затем мы загружаем сериализованную сеть через load_model и добавляем ее в список моделей.

Наконец, мы готовы оценить наш ансамбль:

```
42 # инициализируем список предсказаний 43
print("[INFO] оценивая ансамбль...") 44 предсказания
= []
45
46 # цикл по моделям 47 для
  модели в моделях:
48     # использовать текущую модель для прогнозирования данных тестирования, #
49     # затем сохранить эти прогнозы в списке агрегированных прогнозов
50     predicts.append(model.predict(testX, batch_size=64))
51
52 # усреднить вероятности по всем предсказаниям модели, затем показать 53 # отчет о
  классификации 54 предсказания = np.average(predictions, axis=0) 55
print(classification_report(testY.argmax(axis=1),
56     прогнозы.argmax(ось=1), target_names=labelNames))
```

В строке 44 мы инициализируем наш список прогнозов. Каждая модель в списке моделей будет производить десять вероятностей (по одной для каждой метки класса в наборе данных CIFAR-10) для каждой точки данных в тестировании.

набор. Учитывая, что в наборе данных CIFAR-10 10 000 точек данных, каждая модель будет производить массив размером $10\ 000 \times 10$ — каждая строка соответствует заданной точке данных, а каждый столбец — соответствующая вероятность.

Чтобы накопить эти прогнозы, мы перебираем каждую отдельную модель в строке 47. Затем мы вызываем .predict на данных тестирования и обновить список прогнозов вероятностями, полученными соответствующую модель. После того, как мы перебрали пять моделей в нашем ансамбле и обновили списка прогнозов , наш массив прогнозов теперь имеет форму (5, 10000, 10), что означает, что имеется пять моделей, каждая из которых дает 10 вероятностей меток классов для каждого из 10 000 значений. тестирование точек данных. Затем в строке 54 усредняются вероятности для каждой точки данных тестирования по всем пять моделей.

Чтобы убедиться в этом, мы можем исследовать форму нашего массива прогнозов , который теперь (10000, 10) , подразумевая, что вероятности для каждой из пяти моделей были усреднены вместе. Усреднение — вот почему мы называем этот метод ансамблем — мы берем выход нескольких, независимых моделей и усреднение их вместе для получения окончательного результата. По мнению Дженсена Неравенство, применение методов ансамбля должно работать не хуже (в среднем), чем выбор одного из отдельные модели случайным образом.

Наконец, строки 55 и 56 отображают отчет о классификации для наших предсказаний ансамбля. К определить, повысила ли точность классификации наш ансамбль моделей MiniVGGNet, выполнить следующая команда:

	точность	вспомнить	поддержку	f1-score
самолет	0,89	0,82	0,85	1000
автомобиль	0,93	0,93	0,93	1000
птица	0,80	0,74	0,77	1000
Кот	0,72	0,67	0,69	1000
олень	0,80	0,86	0,83	1000
собака	0,77	0,77	0,77	1000
лягушка	0,86	0,91	0,88	1000
лошадь	0,91	0,87	0,89	1000
корабль	0,89	0,94	0,92	1000
грузовик	0,87	0,93	0,90	1000
среднее / общее	0,84	0,84	0,84	10000

Глядя на отчет о классификации вывода, мы видим, что мы увеличили нашу точность с 83% до 84%, просто объединив выход нескольких сетей, хотя эти сети были обучены на одном и том же наборе данных с использованием одних и тех же гиперпараметров. В общем, можно ожидать повышение точности на 1-5% при применении ансамблей сверточных нейронных сетей в зависимости в вашем наборе данных.

6.2 Резюме

В этой главе мы рассмотрели технику машинного обучения ансамблей и способы обучения нескольких, независимые модели с последующим усреднением результатов могут повысить точность классификации.

Теоретическое обоснование ансамблевых методов можно найти, просмотрев неравенство Дженсена, в котором говорится, что в среднем нам лучше усреднить результаты нескольких моделей вместе , чем выбирать одну наугад.

На самом деле лучшие результаты, которые вы видите в современных статьях (включая Inception [17], ResNet [24] и т. д.), являются средними по некоторым моделям (обычно 3-5, в зависимости от того, как долго авторы должны были обучить свои сети до того, как наступит срок их публикации). В зависимости от вашего набора данных вы обычно можете ожидать увеличения точности на 1-5%.

Хотя ансамбли могут быть простым методом повышения точности классификации, они также требуют больших вычислительных ресурсов — вместо обучения одной сети мы теперь несем ответственность за обучение N из них. Обучение CNN уже является трудоемкой операцией, поэтому метод с линейным масштабированием может быть нецелесообразным в некоторых ситуациях.

Чтобы облегчить вычислительную нагрузку при обучении нескольких моделей, Huang et al. [25] предлагают идею использования циклических графиков скорости обучения для обучения нескольких моделей в течение одного процесса обучения в своей статье 2017 года «Ансамбли моментальных снимков: поезд 1, получите M бесплатно».

Этот метод работает, начиная обучение с высокой скоростью обучения, быстро снижая ее, сохраняя веса модели, а затем сбрасывая скорость обучения до исходного значения без повторной инициализации весов сети. Это действие позволяет сети теоретически расширить покрытие до областей локальных минимумов (или, по крайней мере, областей с низкими потерями) несколько раз в процессе обучения. Ансамбли снимков выходят за рамки этой книги, но их стоит изучить, если вам нужно повысить точность классификации, но вы не можете позволить себе обучать несколько моделей.

7. Расширенные методы оптимизации

До сих пор в этой книге мы изучали и использовали только стохастический градиентный спуск (SGD) для оптимизации наших сетей, но есть и другие методы оптимизации, которые используются в глубоком обучении. В частности, эти более продвинутые методы оптимизации направлены на:

1. Уменьшите количество времени (т. е. количество эпох), чтобы получить разумную классификацию. точность.
2. Сделать сеть более «хорошей» для большего диапазона гиперпараметров, кроме скорости обучения.
3. В идеале получить более высокую точность классификации, чем это возможно с SGD.

С последним воплощением глубокого обучения произошел взрыв новых методов оптимизации, каждый из которых стремится улучшить SGD и обеспечить концепцию адаптивных скоростей обучения. Как мы знаем, SGD изменяет все параметры в сети в равной степени пропорционально заданной скорости обучения. Однако, учитывая, что скорость обучения сети является (1) наиболее важным гиперпараметром для настройки и (2) сложным и утомительным гиперпараметром для правильной настройки, исследователи глубокого обучения предположили, что можно адаптивно настраивать скорость обучения (и в некоторых случаях для каждого параметра) по мере обучения сети.

В этой главе мы рассмотрим методы адаптивной скорости обучения. Я также предложу варианты алгоритмов оптимизации, которые вы должны использовать в своих проектах.

7.1 Методы адаптивной скорости обучения

Чтобы понять каждый из алгоритмов оптимизации в этом разделе, мы рассмотрим их с точки зрения псевдокода, в частности шага обновления. Большая часть этой главы была вдохновлена превосходным обзором методов оптимизации Карпати [26] и Рудером [27]. Мы расширим (а в некоторых случаях и упростим) их объяснения этих методов, чтобы сделать содержание более удобоваримым.

Для начала давайте взглянем на уже знакомый нам алгоритм — этап обновления ванильного SGD:

Здесь у нас есть три значения:

1. W: Наша весовая матрица.
2. lr: Скорость обучения.
3. dW: Градиент W.

Наша скорость обучения здесь фиксирована, и, если она достаточно мала, мы знаем, что наши потери будут уменьшаться во время обучения. Мы также видели расширения SGD, которые включают импульс и ускорение Нестерова в главе 7. Учитывая это обозначение, давайте рассмотрим распространенные оптимизаторы скорости адаптивного обучения, с которыми вы столкнетесь в своей карьере глубокого обучения.

7.1.1 Адаград

Первый метод адаптивной скорости обучения, который мы собираемся изучить, — это Adagrad, впервые представленный Duchi et al [28]. Adagrad адаптирует скорость обучения к параметрам сети. Более крупные обновления выполняются для параметров, которые изменяются нечасто, в то время как меньшие обновления выполняются для часто изменяющихся параметров.

Ниже мы можем увидеть представление псевдокода обновления Adagrad:

```
кеш += (dW ** 2)
W += -lr * dW/(np.sqrt(кеш) + eps)
```

Первый параметр, который вы заметите, — это кеш — эта переменная поддерживает сумму квадратов градиентов для каждого параметра и обновляется в каждом мини-пакете в процессе обучения. Исследуя кеш, мы можем увидеть, какие параметры обновляются часто, а какие редко.

Затем мы можем разделить lr * dx на квадратный корень из кеша (добавив значение эпсилон для сглаживания и предотвращения ошибок деления на ноль). Масштабирование обновления по всей предыдущей сумме квадратных градиентов позволяет нам адаптивно обновлять параметры в нашей сети.

Веса, которые часто обновляются/большие градиенты в кеше, уменьшают размер обновления, эффективно снижая скорость обучения для параметра. С другой стороны, веса, которые имеют нечастые обновления/меньшие градиенты в кеше, будут увеличивать размер обновления, эффективно повышая скорость обучения для конкретного параметра.

Основное преимущество Adagrad заключается в том, что нам больше не нужно вручную настраивать скорость обучения — большинство реализаций алгоритма Adagrad оставляют начальную скорость обучения на уровне 0,01 и позволяют адаптивному характеру алгоритма настраивать скорость обучения для каждого параметра.

Однако слабость Адаграда можно увидеть, изучив тайник. В каждой мини-партии квадраты градиентов накапливаются в знаменателе. Поскольку градиенты возводятся в квадрат (и, следовательно, всегда положительны), это накопление продолжает расти и расти в процессе обучения. Как мы знаем, деление небольшого числа (градиента) на очень большое число (кеш) приведет к бесконечно малому обновлению, слишком малому для того, чтобы сеть действительно могла что-то узнать в более поздние эпохи.

Это явление происходит даже для небольших, редко обновляемых параметров, поскольку положительные значения в кеше монотонно растут, поэтому мы редко видим, как Adagrad используется для обучения (современных) нейронных сетей глубокого обучения. Тем не менее, важно просмотреть, чтобы мы могли понять расширения алгоритма Adagrad.

7.1.2 Ададельта

Алгоритм Ададельта был предложен Зейлером в их статье 2012 года ADADELTA: An Adaptive Learning Rate Method [29]. Adadelta можно рассматривать как расширение Adagrad, которое направлено на уменьшение монотонно уменьшающейся скорости обучения, вызванной кешем.

В алгоритме Adagrad мы обновляем наш кеш всеми ранее возведенными в квадрат градиентами. Однако Adadelta ограничивает это обновление кеша, накапливая только небольшое количество прошлых градиентов — при фактической реализации эта операция сводится к вычислению убывающего среднего значения всех прошлых квадратов градиентов.

Таким образом, Ададельту можно рассматривать как улучшение Адаграда; однако очень тесно связаны Алгоритм RMSprop (который также выполняет расщепление кэша) часто предпочтительнее, чем Adadelta.

7.1.3 RMSprop

Алгоритм RMSprop, разработанный независимо от Adadelta, представляет собой (неопубликованный) алгоритм оптимизации, показанный на слайдах курса Джейфри Хинтона на Coursera [2]. Подобно Adadelta, RMSprop пытается устранить негативные последствия глобально накопленного кеша путем преобразования кеша в экспоненциально взвешенное скользящее среднее.

Давайте посмотрим на обновление псевдокода RMSprop:

```
кеш = скорость_затухания * кеш + (1 - скорость_затухания) * (dW ** 2)
W += -lr * dW / (np.sqrt(кеш) + eps)
```

Первый аспект RMSprop, который вы заметите, заключается в том, что фактическое обновление весовой матрицы W идентично обновлению Adagrad — здесь важно то, как обновляется кеш . Скорость распада, часто определяемая как ρ , представляет собой гиперпараметр, обычно равный 0,9. Здесь мы видим, что предыдущие записи в кеше будут иметь значительно меньший вес, чем новые обновления. Этот аспект RMSprop со «скользящим средним» позволяет кешу «выводить» старые квадратичные градиенты и заменять их более новыми, «свежими».

Опять же, фактическое обновление W идентично обновлению Adagrad — суть алгоритма зависит от экспоненциального уменьшения кеша, что позволяет нам избежать монотонного снижения скорости обучения в процессе обучения. На практике RMSprop имеет тенденцию быть более эффективным, чем Adagrad и Adadelta, когда применяется для обучения различных сетей глубокого обучения [5]. Кроме того, RMSprop имеет тенденцию сходиться значительно быстрее, чем SGD.

Помимо SGD, RMSprop, возможно, был вторым наиболее часто используемым алгоритмом оптимизации в недавней литературе по глубокому обучению; однако следующий метод оптимизации, который мы собираемся обсудить, Adam, сейчас используется чаще, чем RMSprop.

7.1.4 Адам

Алгоритм оптимизации Адама (адаптивная оценка момента), предложенный Кингмой и Ба в их статье 2014 года «Адам: метод стохастической оптимизации» [1], по сути является RMSprop, только с добавленным к нему импульсом:

```
m = beta1 * m + (1 - beta1) * dW v = beta2
* v + (1 - beta2) * (dW ** 2) x += -lr * m / (np.sqrt(v)
+ eps)
```

 Опять же, я хочу привлечь особое внимание к этим обновлениям псевдокода, поскольку они были получены и популяризированы благодаря превосходным заметкам о методах оптимизации Карпати [26].

Значения как m , так и v аналогичны импульсу SGD, исходя из их соответствующих предыдущих значений с момента времени $t-1$. Значение m представляет первый момент (среднее значение) градиентов, а v — второй момент (дисперсию).

Фактическое обновление W почти идентично RMSprop, только теперь мы используем «сглаженную» версию (из-за вычисления среднего) m , а не необработанный градиент dW — использование среднего приводит к более желательным обновлениям, насколько мы можем сгладить зашумленные обновления необработанных значений dW . Обычно beta1 устанавливается на 0,9, а beta2 — на 0,999 — эти значения редко (если вообще) изменяются при использовании оптимизатора Adam.

На практике во многих ситуациях Adam работает лучше, чем RMSprop. Для получения более подробной информации о алгоритме оптимизации Адама см. Kingma and Ba [1].

7.1.5 Надам

Точно так же, как Adam — это RMSprop с импульсом, Надам — это RMSprop с ускорением Нестерова.

Надам был предложен Тимоти Дозатом, доктором философии. студент Стэнфордского университета [30]. Обычно мы не видим, чтобы слово «Надам» использовалось «в дикой природе», но важно понимать, что этот вариант слова «Адам» действительно существует.

7.2 Выбор метода оптимизации

Учитывая выбор между всеми этими алгоритмами оптимизации, какой из них выбрать?

К сожалению, ответ весьма неубедителен — работа Шаула и соавт. в 2014 году модульные тесты для стохастической оптимизации [31] попытались сравнить многие из этих методов оптимизации и обнаружили, что, хотя алгоритмы адаптивной скорости обучения работают хорошо, явного победителя не было.

Алгоритмы оптимизации глубокого обучения (и способы их выбора) по-прежнему остаются открытой областью исследований и, вероятно, останутся таковыми в течение многих лет. Поэтому вместо того, чтобы исчерпывающе пробовать каждый алгоритм оптимизации, который вы можете найти, бросая каждый в свой набор данных и отмечая, что прилипает, лучше освоить два или три алгоритма оптимизации. Часто успех проекта глубокого обучения зависит от сочетания алгоритма оптимизации (и связанных с ним параметров) и того, насколько искусно исследователь «управляет» алгоритмом.

7.2.1 Три метода обучения вождению: SGD, Adam и RMSprop

«Выбор того, какой алгоритм использовать, на данный момент, по-видимому, во многом зависит от знакомства пользователя с алгоритмом (для простоты настройки гиперпараметров)».
— Гудфеллоу и др. [5]

Учитывая успех алгоритмов адаптивной скорости обучения, таких как RMSprop и Adam, у вас может возникнуть соблазн просто игнорировать SGD и относиться к нему как к архаичному инструменту. В конце концов, существуют «лучшие» методы, не так ли?

Однако подразумевать игнорирование SGD было бы большой ошибкой. Взгляните на любую недавнюю современную публикацию по глубокому обучению сложных наборов данных классификации изображений, таких как ImageNet: AlexNet [6], VGGNet [11], SqueezeNet [32], Inception [17], ResNet [33] — все из этих современных архитектур были обучены с использованием SGD.

Но почему это? Мы ясно видим преимущества алгоритмов с адаптивной скоростью обучения, таких как RMSprop и Adam, — сети могут сходиться быстрее. Однако скорость сходимости хоть и важна, но не является самым важным фактором — гиперпараметры все же побеждают. Если вы не можете настроить гиперпараметры для данного оптимизатора (и связанной с ним модели), ваша сеть никогда не достигнет приемлемой точности.

Хотя SGD, безусловно, сходится медленнее, чем алгоритмы адаптивной скорости обучения, это также более изученный алгоритм. Исследователи лучше знакомы с SGD и годами используют его для обучения сетей.

Возьмем, к примеру, профессионального автогонщика, который в течение пяти лет водит гоночную машину одной и той же марки и модели. Затем, в один прекрасный день, спонсор водителя меняется, и они вынуждены

управлять новым транспортным средством. У гонщика нет времени опробовать новую гоночную машину, и он вынужден начать гонку, не имея опыта вождения автомобиля. Будет ли гонщик так же хорошо выступать в своих первых гонках? Скорее всего нет — водитель не знаком с транспортным средством и его тонкостями (но все же может действовать разумно, поскольку водитель все-таки профессионал).

То же самое касается архитектур глубокого обучения и алгоритмов оптимизации. Чем больше экспериментов мы проводим с заданной архитектурой и алгоритмом оптимизации, тем больше мы узнаем о тонкостях процесса обучения. Учитывая, что SGD был краеугольным камнем обучения нейронных сетей в течение почти 60 лет, неудивительно, что этот алгоритм все еще постоянно используется сегодня — скорость сходимости просто не имеет значения (так сильно) по сравнению с производительностью (точностью) алгоритма . модель.

Прошу говоря: если мы сможем получить более высокую точность для заданного набора данных с помощью SGD, мы, вероятно, будем использовать SGD, даже если обучение займет в 1,5 раза больше времени, чем при использовании Adam или RMSprop, просто потому, что мы лучше понимаем гиперпараметры. В настоящее время наиболее часто используемыми алгоритмами оптимизации глубокого обучения являются: 1. SGD 2. RMSprop 3. Adam

Я бы порекомендовал вам сначала освоить SGD и применять его к каждой архитектуре и набору данных, с которыми вы сталкиваетесь. В некоторых случаях. он будет работать отлично, а в других - плохо. Цель здесь состоит в том, чтобы вы столкнулись с как можно большим количеством проблем глубокого обучения, используя определенный алгоритм оптимизации, и научились настраивать связанные гиперпараметры. Помните, что глубокое обучение — это отчасти наука и отчасти искусство — овладение алгоритмом оптимизации — это абсолютно искусство , требующее большой практики. Оттуда перейдите либо к RMSprop, либо к Adam.

Я лично рекомендую изучить Адама до RMSprop, поскольку, по моему опыту, Адам склонен чтобы превзойти RMSprop в большинстве ситуаций.

7.3 Резюме

В этой главе мы обсудили адаптивные алгоритмы оптимизации скорости обучения, которые можно использовать вместо SGD. Выбор алгоритма оптимизации для обучения глубокой нейронной сети во многом зависит от вашего знакомства с: 1. Набором данных.

2. Архитектура модели. 3.

Алгоритм оптимизации (и связанные с ним гиперпараметры).

Вместо того, чтобы усердно проводить эксперименты, чтобы попробовать каждый алгоритм оптимизации, который вы можете найти, вместо этого лучше освоить два или три метода и то, как настроить их гиперпараметры. Становясь экспертом в этих методах, вы сможете с гораздо большей легкостью применять новые архитектуры моделей к наборам данных, с которыми вы раньше не работали.

Моя личная рекомендация — посвятить много времени в начале своей карьеры глубокому обучению тому, как использовать SGD; в частности, SGD с импульсом. Как только вы почувствуете себя комфортно, применяя SGD к различным архитектурам и наборам данных, переходите к Adam и RMSprop.

Наконец, имейте в виду, что скорость сходимости скорости модели является вторичной по отношению к потерям и точности — выберите алгоритм оптимизации, на который вы можете (уверенно) настроить гиперпараметры, что приведет к разумно эффективной сети.

8. Оптимальный путь для применения глубокого обучения

В главе 10 Starter Bundle мы рассмотрели рецепт обучения нейронной сети. Четыре ингредиента рецепта включали:

1. Ваш набор данных
2. Функция потерь 3.
- Архитектура нейронной сети 4. Метод оптимизации Используя этот рецепт, мы

можем обучить любой тип модели глубокого обучения. Тем не менее, этот рецепт не охватывает оптимальный способ объединения этих ингредиентов вместе, а также то, с какими частями рецепта вам нужно возиться, если вы не получаете желаемых результатов.

Как вы узнаете в своей карьере в области глубокого обучения, возможно, самым сложным аспектом глубокого обучения является изучение вашей кривой точность/потери и принятие решения о том, что делать дальше. Если ваша ошибка обучения слишком высока, что вы делаете? Что произойдет, если ваша ошибка проверки также высока? Как вы корректируете свой рецепт, когда ваша ошибка проверки соответствует вашей ошибке обучения или ошибки контабраздатов высока?

В этой главе я расскажу об оптимальном способе применения методов глубокого обучения, начиная с эмпирических правил, которые вы можете использовать для корректировки своего рецепта обучения. Затем я предоставлю процесс принятия решения, который вы можете использовать при принятии решения о том, следует ли вам обучать свою модель глубокого обучения с нуля или применять трансферное обучение. К концу этой главы вы будете хорошо понимать эмпирические правила, которые практикующие специалисты по глубокому обучению используют при обучении своих собственных сетей.

8.1 Рецепт обучения

Следующий раздел в значительной степени вдохновлен превосходным учебным пособием Эндрю Нг на NIPS 2016 под названием «Гайки и болты создания приложений глубокого обучения» [34]. В этом выступлении Нг обсудил, как мы можем заставить методы глубокого обучения работать в наших собственных продуктах, бизнесе и академических исследованиях. Возможно, самый важный вывод из выступления Нг следует (резюмировано Малисевичем [35]):

 «Большинство проблем в прикладном глубоком обучении возникает из-за несоответствия данных обучения/тестирования. В некоторых сценариях эта проблема просто не возникает, но вы будете удивлены, как часто применяется ма-

В китайских учебных проектах используются обучающие данные (которые легко собирать и аннотировать), которые отличаются от целевого приложения». - Эндрю Нг (резюмировано Малисевичем)

И Нг, и Малисевич говорят здесь о том, что вы должны проявлять мучительную осторожность, чтобы убедиться, что ваши обучающие данные репрезентативны для ваших наборов проверки и тестирования. Да, получение, аннотирование и маркировка набора данных требует очень много времени и даже в некоторых случаях очень дорого. И да, методы глубокого обучения имеют тенденцию хорошо обобщать в определенных ситуациях. Однако нельзя ожидать, что какая-либо модель машинного обучения, обученная на нерепрезентативных данных, будет успешной.

Например, предположим, что нам поручили создать систему глубокого обучения, отвечающую за распознавание марки и модели автомобиля с камеры, установленной на нашем автомобиле, когда мы едем по дороге (рис. 8.1, слева).



Рисунок 8.1: Слева: автомобили на шоссе, которые мы хотим идентифицировать с помощью глубокого обучения. Справа: пример изображения «снимка продукта», на котором фактически обучалась наша сеть.

Первым шагом является сбор наших обучающих данных. Чтобы ускорить процесс сбора данных, мы решили собрать веб-сайты, на которых есть как фотографии автомобилей, так и их марка и модель, указанные на веб-странице — отличные примеры таких веб-сайтов включают Autotrader.com, eBay, CarMax и т. д. Для каждого из этих веб-сайтов, мы можем создать простого паука, который сканирует веб-сайт, находит списки отдельных продуктов (т. е. «страницы автомобилей», на которых перечислены характеристики автомобиля), а затем загружает изображения и производит + информацию о модели.

Этот метод довольно прост, и помимо времени, которое требуется нам на разработку паука, нам не потребуется много времени, чтобы накопить достаточно большой размеченный набор данных. Затем мы разделяем этот набор данных на два: обучение и проверку, и приступаем к обучению данной архитектуры глубокого обучения с высокой точностью (> 90%).

Однако когда мы применяем нашу недавно обученную модель к примерным изображениям, таким как на рис. 8.1 (слева), мы обнаруживаем, что результаты ужасны — нам повезло получить 5-процентную точность при развертывании в реальном мире. Почему это?

Причина в том, что мы выбрали легкий путь. Мы не остановились на том, что снимки автомобилей, представленные на Autotrader, CarMax и eBay (рис. 8.1, справа), не являются репрезентативными для автомобилей, которые наша система машинного зрения будет видеть прикрепленными к приборной панели нашего автомобиля. Хотя наша система глубокого обучения может отлично определять марку и модель автомобиля на снимке продукта, она не сможет распознать марку модели автомобиля ни спереди, ни сзади, как это часто бывает при вождении.

Нет быстрого пути к созданию собственного набора данных изображений. Если вы ожидаете, что система глубокого обучения получит высокую точность в данной реальной ситуации, убедитесь, что эта система глубокого обучения была обучена на изображениях, характерных для того места, где она будет развернута, иначе вы будете очень разочарованы ее производительностью.

8.1 Рецепт обучения

Предполагая, что мы собрали достаточно обучающих данных, которые представляют задачу классификации, которую мы пытаемся решить, Эндрю Нг предоставил четырехэтапный процесс, чтобы помочь нам в нашем обучении [34].

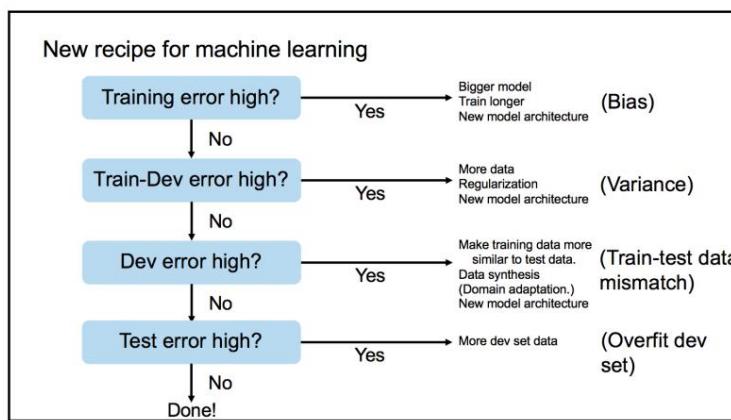


Рисунок 8.2: Слайд 13 выступления Эндрю Нг [34]. Здесь Нг предлагает четыре отдельных разделения данных при обучении модели глубокого обучения.

На рис. 8.2 видно, что Ng предлагает четыре набора разбиений данных при обучении модели глубокого обучения: 1. Обучение 2. Обучение-валидация (которое Ng называет «разработкой»).

3. Валидация

4. Тестирование

Мы уже видели сплиты обучения, валидации и тестирования, но что это за новый набор «валидации обучения»? Ng рекомендует взять все наши данные и разделить их на 60% для обучения и оставшиеся 40% для тестирования. Затем мы разделяем данные тестирования на две части: одну для проверки, а другую для истинного тестирования (то есть данные, которые мы никогда не трогаем, пока не будем готовы оценить производительность нашей сети). Затем из нашего тренировочного набора мы берем небольшой фрагмент и добавляем его в наш «тренировочно-проверочный набор». Учебный набор поможет нам определить смещение нашей модели, а набор для обучения и проверки поможет определить дисперсию.

Если наша ошибка обучения слишком велика, как показано на рис. 8.3 (вверху слева) ниже, то нам следует рассмотреть возможность углубления нашей текущей архитектуры путем добавления дополнительных слоев и нейронов. Мы также должны рассмотреть возможность обучения в течение более длительного времени (т. е. большего количества эпох), одновременно настраивая нашу скорость обучения — использование меньшей скорости обучения может позволить вам тренироваться дольше, помогая предотвратить переобучение. Наконец, если после многих экспериментов с использованием нашей текущей архитектуры и различных скоростей обучения она не окажется полезной, нам, вероятно, придется попробовать совершенно другую архитектуру модели.

Переходя ко второму пункту на блок-схеме, если наша ошибка обучения-валидации высока (рис. 8.3, вверху справа), то мы должны проверить параметры регуляризации в нашей сети. Применяем ли мы отсевающие слои внутри сетевой архитектуры? Используется ли дополнение данных для создания новых обучающих выборок? А как насчет самой функции потери/обновления — включены ли штрафы за регуляризацию? Изучите эти вопросы в контексте ваших собственных экспериментов по глубокому обучению и начните добавлять регуляризацию.

Вам также следует подумать о сборе большего количества обучающих данных (опять же, позаботившись о том, чтобы эти обучающие данные представляли то, где будет развернута модель) на этом этапе — почти во всех случаях наличие большего количества обучающих данных никогда не бывает плохим. Вполне вероятно, что у вашей модели недостаточно данных для обучения, чтобы изучить основные закономерности в ваших примерах изображений. Наконец, исчерпав эти варианты,

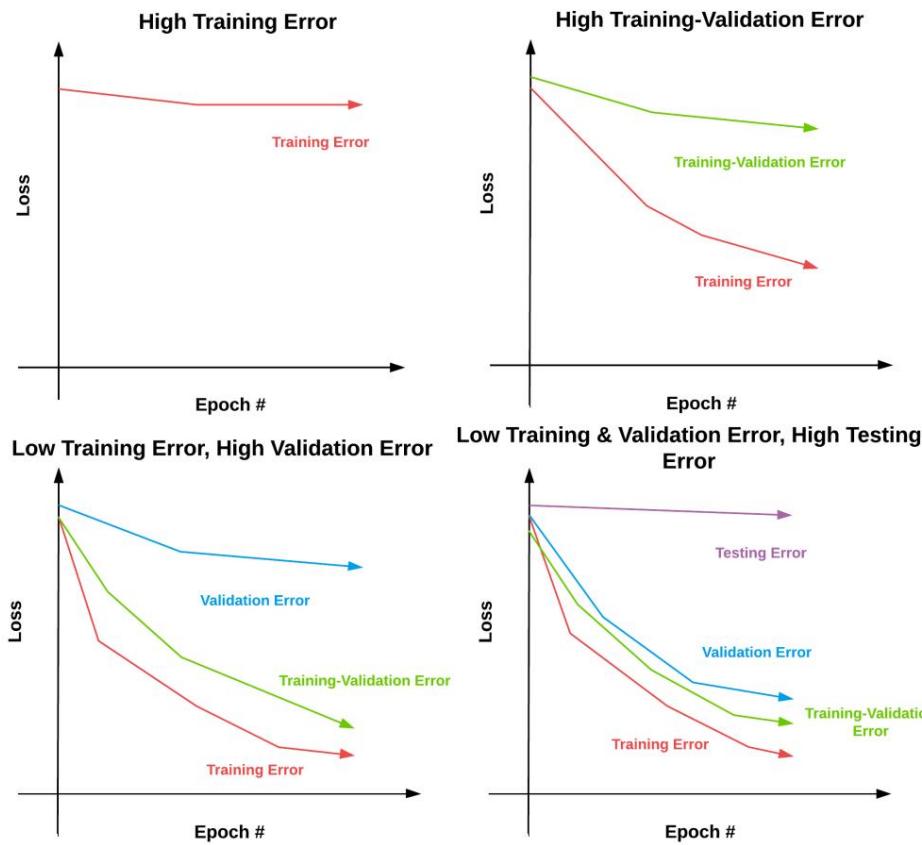


Рисунок 8.3: Четыре этапа рецепта машинного обучения Эндрю Нг. Вверху справа: наша ошибка обучения высока, что означает, что нам нужна более мощная модель для представления основных закономерностей в данных. Вверху слева: наша ошибка обучения уменьшилась, но наша ошибка проверки обучения высока. Это означает, что мы должны получить больше данных или применить сильную регуляризацию. Внизу слева: если и ошибка обучения , и ошибка проверки обучения малы, но ошибка проверки высока, мы должны проверить наши данные обучения и убедиться, что они правильно имитируют наши наборы проверки и тестирования. Внизу справа: если обучение, обучение-валидация и ошибка проверки низкие, но ошибка тестирования высока, нам нужно собрать больше данных обучения + проверки.

вы снова захотите рассмотреть возможность использования другой сетевой архитектуры.

Продолжая рассматривать блок-схему на рисунке 8.3 (внизу слева), если наша ошибка обучения-валидации низкая, но наша ошибка набора проверки высока, нам нужно более внимательно изучить наши обучающие данные. Мы абсолютно уверены, что наши тренировочные изображения похожи на наши проверочные изображения?

Будьте честны с собой — вы не можете ожидать, что модель глубокого обучения, обученная на изображениях, не представляющих изображения, которые они увидят в условиях проверки или тестирования, будет работать хорошо. Если вы с трудом осознаете, что это действительно так, вернитесь к этапу сбора набора данных и потратите время на сбор дополнительных данных. Без данных, представляющих, где будет развернута ваша модель глубокого обучения, вы не получите высокоточных результатов. Вам также следует еще раз проверить параметры регуляризации — достаточно ли сильно вы регуляризируете? Наконец, вам может снова понадобиться рассмотреть новую архитектуру модели.

Наконец, мы переходим к последнему шагу в блок-схеме — велика ли наша ошибка тестирования? На данный момент мы подогнали нашу модель к данным обучения и проверки (рис. 8.3, справа внизу). Нам нужно вернуться и собрать больше данных для набора проверки, чтобы помочь нам определить, когда это переоснащение

начинает происходить. Используя эту методологию, предложенную Эндрю Нг, мы можем легче принимать (правильные) решения относительно обновления нашей модели/набора данных, когда наши эксперименты не окажутся такими, как мы ожидали.

8.2 Перенос обучения или обучение с нуля

Следующий раздел вдохновлен прекрасным уроком «Перенос обучения» в Стэнфордском классе cs231n [36]. Я также включил свой собственный анекдотический опыт, чтобы помочь вам в ваших собственных экспериментах. Учитывая успех трансферного обучения в главе 3 об извлечении признаков и главе 5 о тонкой настройке, вы можете задаться вопросом, когда следует применять трансферное обучение, а когда — обучать модель с нуля.

Чтобы принять это решение, вам необходимо учитывать два важных фактора: 1.

Размер вашего набора данных.

2. Сходство вашего набора данных с набором данных, на котором обучалась предварительно обученная CNN (обычно это ImageNet).

На основе этих факторов мы можем построить диаграмму, которая поможет нам принять решение о том, нужно ли нам применять трансферное обучение или обучать с нуля (рис. 8.4). Давайте рассмотрим каждую из четырех возможностей ниже.

	Similar Dataset	Different Dataset
Small Dataset	Feature extraction using FC layers + classifier	Feature extraction using lower level CONV layers + classifier
Large Dataset	Fine-tuning likely to work, but might have to train from scratch	Fine-tuning worth trying, but will likely not work; likely have to train from scratch

Рисунок 8.4: Таблица, которую вы можете использовать, чтобы определить, следует ли вам обучать нашу сеть с нуля или передавать обучение. Рисунок вдохновлен Грегом Чу из Deep Learning Sandbox [37].

Ваш набор данных мал и подобен исходному набору данных. Поскольку ваш набор данных невелик, у вас, вероятно, недостаточно обучающих примеров для обучения CNN с нуля (опять же, имейте в виду, что в идеале у вас должно быть 1000-5000 примеров на класс, который вы хотите распознавать). Кроме того, учитывая отсутствие данных для обучения, вероятно, не стоит пытаться выполнить точную настройку, поскольку мы, вероятно, в конечном итоге получим переобучение.

Вместо этого, поскольку ваш набор данных изображения аналогичен тому, на котором обучалась предварительно обученная сеть, вы должны рассматривать сеть как экстрактор признаков и обучать простой классификатор машинного обучения поверх этих признаков. Вы должны извлекать признаки из слоев, расположенных глубже в архитектуре, так как эти признаки являются более богатыми и представляют шаблоны, извлеченные из исходного набора данных.

Ваш набор данных большой и подобен исходному набору данных. С большим набором данных у нас должно быть достаточно примеров, чтобы применить точную настройку без переобучения. У вас может возникнуть соблазн обучить свою собственную модель с нуля — это эксперимент, который стоит провести. Однако, поскольку ваш набор данных подобен исходному набору данных, на котором сеть уже обучалась, фильтры внутри сети, вероятно, уже достаточно различительны, чтобы получить разумный классификатор. Поэтому применяйте тонкую настройку в этом случае.

Ваш набор данных мал и отличается от исходного набора данных. Опять же, учитывая небольшой набор данных, мы, вероятно, не получим высокоточную модель глубокого обучения, обучаясь с нуля. Вместо этого мы должны снова применить извлечение признаков и обучить стандартный

модель машинного обучения поверх них, но, поскольку наши данные отличаются от исходного набора данных, мы должны использовать слои более низкого уровня в сети в качестве наших экстракторов признаков.

Имейте в виду, что чем глубже мы погружаемся в сетевую архитектуру, тем более богатыми и разборчивыми являются нативные функции, специфичные для набора данных, на котором он обучался. Извлекая признаки из нижних слоев сети, мы по-прежнему можем использовать эти фильтры, но без абстракции, вызванной более глубокими слоями.

Ваш новый набор данных большой и отличается от исходного набора данных . В этом случае у нас есть два варианта. Учитывая, что у нас достаточно обучающих данных, мы, вероятно, сможем обучить нашу собственную сеть с нуля. Однако предварительно обученные веса из моделей, обученных на наборе данных, таком как ImageNet, обеспечивают превосходную инициализацию, даже если наборы данных не связаны. Поэтому мы должны выполнить два набора экспериментов: 1. В первом наборе экспериментов попытаться точно настроить предварительно обученную сеть на ваш набор данных и оценить производительность.

2. Затем во второй серии экспериментов обучите совершенно новую модель с нуля и оцените.

Какой именно метод работает лучше всего, полностью зависит от вашего набора данных и проблемы классификации. Тем не менее, я бы порекомендовал сначала попытаться выполнить точную настройку , так как этот метод позволит вам установить базовый уровень, который будет превзойден, когда вы перейдете ко второму набору экспериментов и обучите свою сеть с нуля.

8.3 Резюме

В этой главе мы рассмотрели оптимальный путь применения методов глубокого обучения при обучении собственных сетей. При сборе обучающих данных помните, что ярлыков не существует — найдите время, чтобы убедиться, что данные, которые вы используете для обучения своей модели, представляют изображения, которые ваша сеть увидит при развертывании в реальном приложении.

Есть старый информационный анекдот, который гласит: «Мусор на входе, мусор на выходе». Если ваши входные данные не представляют собой примеры точек данных, которые ваша модель увидит после обучения, вы, по сути, попадаете в эту ловушку «мусор на входе — мусор на выходе». Это не значит, что ваши данные — «мусор». Вместо этого напомните себе об этом анекдоте при проведении собственных экспериментов и поймите, что ваша модель глубокого обучения не может хорошо работать с точками данных, которые она никогда не обучала распознавать.

Мы также рассмотрели, когда вам следует подумать о переносе обучения по сравнению с обучением собственной сети с нуля. С небольшими наборами данных вам следует подумать об извлечении признаков. Для больших наборов данных сначала рассмотрите точную настройку (чтобы установить базовый уровень), а затем перейдите к обучению модели с нуля.

9. Работа с HDF5 и большими наборами данных

До сих пор в этой книге мы работали только с наборами данных, которые могут поместиться в основную память наших машин. Для небольших наборов данных это разумное предположение — мы просто загружаем каждое отдельное изображение, предварительно обрабатываем его и позволяем передавать его через нашу сеть. Однако для крупномасштабных наборов данных глубокого обучения (например, ImageNet) нам необходимо создать генераторы данных, которые одновременно обращаются только к части набора данных (т. е. к мини-пакету), а затем разрешают передачу пакета по сети. .

К счастью, Keras поставляется с методами, позволяющими использовать необработанные пути к файлам на диске в качестве входных данных для процесса обучения. Вам не нужно хранить весь набор данных в памяти — просто укажите пути к изображениям генератору данных Keras, и ваши изображения будут загружаться партиями и передаваться по сети.

Однако этот метод ужасно неэффективен. Каждое изображение, находящееся на вашем диске, требует операции ввода-вывода, которая вносит задержку в конвейер обучения. Обучение сетей глубокого обучения уже достаточно медленное — нам бы не мешало максимально избежать узкого места ввода-вывода .

Более элегантным решением было бы сгенерировать набор данных HDF5 для ваших необработанных изображений, как мы это делали в главе 3 о переносе обучения и извлечении признаков, только на этот раз мы сохраняем сами изображения, а не извлеченные признаки. HDF5 не только способен хранить массивные наборы данных, но и оптимизирован для операций ввода-вывода, особенно для извлечения пакетов (называемых «срезами») из файла. Как мы увидим в оставшейся части этой книги, дополнительный шаг по упаковке необработанных изображений, находящихся на диске, в файл HDF5 позволяет нам создать структуру глубокого обучения , которую можно использовать для быстрого создания наборов данных и обучения сетей глубокого обучения на верх из них.

В оставшейся части этой главы я покажу, как создать набор данных HDF5 для соревнования Kaggle Dogs vs. Cats [3]. Затем, в следующей главе, мы будем использовать этот набор данных HDF5 для обучения исходной архитектуры AlexNet [6], что в конечном итоге приведет к 25-й позиции в таблице лидеров в следующей главе.

9.1 Загрузка Kaggle: Dogs vs. Cats

Чтобы загрузить набор данных Kaggle: Dogs vs. Cats, вам сначала необходимо создать учетную запись на kaggle.com. Оттуда перейдите на домашнюю страницу Dogs vs. Cats (<http://pyimg.co/xb5lb>).

Вам нужно будет скачать train.zip. Не скачивайте test1.zip. Изображения внутри test1.zip используются только для расчета прогнозов и отправки на оценку Kaggle. сервер. Поскольку нам нужны метки классов для создания наших собственных тренировочных и тестовых сплитов, нам нужно только поезд.zip. Предоставление собственных предсказанных результатов выходит за рамки этой книги, но может быть легко можно выполнить, записав свои прогнозы на test1.zip в формате файла, указанном в образец представления.csv.

После загрузки train.zip разархивируйте его, и вы найдете каталог с именем train — этот каталог содержит наши фактические изображения. Сами ярлыки могут быть получены в результате изучения имен файлов. Я включил образец имен файлов ниже:

```
kaggle_dogs_vs_cats/train/cat.11866.jpg
...
kaggle_dogs_vs_cats/train/dog.11046.jpg
```

Как я рекомендовал в Starter Bundle, я буду использовать следующую структуру данных для этого проекта:

```
|--- наборы данных
| |--- kaggle_dogs_vs_cats
| | |--- hdf5
| | |--- поезд
| --- dog_vs_cats
| |--- конфигурация
| |--- build_dogs_vs_cats.py
| |--- ...
```

Обратите внимание, что я буду хранить каталог поезда , содержащий наши примеры изображений, в папке, предназначеннной исключительно для соревнования Kaggle: Dogs vs. Cats. Оттуда у меня есть dogs_vs_cats каталог, в котором мы будем хранить код для этого проекта.

Теперь, когда мы загрузили набор данных Dogs vs. Cats и изучили нашу структуру каталогов, давайте создадим наш файл конфигурации.

9.2 Создание файла конфигурации

Теперь, когда мы начинаем создавать более продвинутые проекты и методы глубокого обучения, мне нравится создайте специальный модуль конфигурации Python для каждого из моих проектов. Например, вот каталог структура проекта Kaggle Dogs vs. Cats:

```
--- собаки_vs_котов
| |--- конфигурация
| | |--- __init__.py
| | |--- dogs_vs_cats_config.py
| |--- build_dogs_vs_cats.py
| |--- crop_accuracy.py
| |--- Extract_features.py
| |--- train_alexnet.py
| |--- train_model.py
| |--- вывод/
| | |--- __init__.py
| | |--- alexnet_dogs_vs_cats.model
```

```

        |--- dogs_vs_cats_features.hdf5 |---
        |--- dogs_vs_cats_mean.json |---
        |--- dogs_vs_cats.pickle

```

Вы можете пока игнорировать настоящие сценарии Python, так как мы рассмотрим их в следующей главе, но взгляните на каталог с именем config. Внутри конфигурации вы найдете один файл Python с именем dogs_vs_cats_config.py — я использую этот файл для хранения всех соответствующих конфигураций для проекта, включая: 1. Пути к входным изображениям.

2. Общее количество меток класса.
3. Информация о тренировочных, проверочных и тестовых сплитах.
4. Пути к наборам данных HDF5.
5. Пути к выходным моделям, графикам, логам и т.д.

Использование файла Python, а не файла JSON, позволяет мне включать фрагменты кода Python и делает файл конфигурации более эффективным для работы (отличным примером является управление путями к файлам с помощью модуля os.path). Я бы посоветовал вам привыкнуть использовать файлы конфигурации на основе Python для ваших собственных проектов глубокого обучения, поскольку это значительно повысит вашу производительность и позволит вам контролировать большинство параметров в вашем проекте через один файл.

9.2.1 Ваш первый файл конфигурации

Давайте продолжим и посмотрим на мой файл конфигурации (dogs_vs_cats_config.py) для набора данных Kaggle Dogs vs. Cats:

```

1 # определяем пути к каталогу изображений 2 IMAGES_PATH
= "../datasets/kaggle_dogs_vs_cats/train"
3
4 # так как у нас нет данных валидации или доступа к тестовым 5 # меткам нам нужно взять некоторое
количество изображений из обучающих 6 # данных и использовать их вместо них
7 NUM_CLASSES = 2
8 NUM_VAL_IMAGES = 1250 * NUM_CLASSES
9 NUM_TEST_IMAGES = 1250 * NUM_CLASSES
10
11 # определить путь к выходному обучению, проверке и тестированию
12 # файлы HDF5
13 TRAIN_HDF5 = "../наборы данных/kaggle_dogs_vs_cats/hdf5/train.hdf5"
14 VAL_HDF5 = "../наборы данных/kaggle_dogs_vs_cats/hdf5/val.hdf5"
15 TEST_HDF5 = "../наборы данных/kaggle_dogs_vs_cats/hdf5/test.hdf5"

```

В строке 2 я определяю путь к каталогу, содержащему изображения собак и кошек — это изображения, которые мы будем упаковывать в набор данных HDF5 позже в этой главе. Строки 7-9 определяют общее количество меток класса (две: одна для собак, другая для кошки) вместе с количеством проверочных и тестовых изображений (по 2500 для каждого). Затем мы можем указать путь к нашим выходным файлам HDF5 для обучения, проверки и тестирования соответственно в строках 13-15.

Вторая половина файла конфигурации определяет путь к выходным сериализованным весам, среднее значение набора данных и общий «выходной» путь для хранения графиков, отчетов о классификации, журналов и т. д.:

```

17 # путь к выходному файлу модели 18
MODEL_PATH = "output/alexnet_dogs_vs_cats.model"

```

```

20 # определяем путь к набору данных 21
DATASET_MEAN = "output/dogs_vs_cats_mean.json"
22
23 # определить путь к выходному каталогу, используемому для хранения графиков, 24 # отчетов о классификации и т. д.
25 OUTPUT_PATH = "выход"

```

Файл DATASET_MEAN будет использоваться для хранения средних значений интенсивности красных, зеленых и синих пикселей по всему (обучающему) набору данных. Когда мы обучаем нашу сеть, мы будем вычитать средние значения RGB из каждого пикселя изображения (то же самое касается тестирования и оценки). Этот метод, называемый вычитанием среднего, представляет собой тип метода нормализации данных и используется чаще, чем масштабирование интенсивности пикселей до диапазона [0,1], поскольку было показано, что он более эффективен для больших наборов данных и более глубоких нейронных сетей.

9.3 Создание набора данных

Теперь, когда наш файл конфигурации определен, давайте перейдем к созданию наших наборов данных HDF5. Откройте новый файл, назовите его build_dogs_vs_cats.py и вставьте следующий код:

```

1 # импортировать необходимые пакеты
2 из конфига import dogs_vs_cats_config as config 3 из
sklearn.preprocessing import LabelEncoder 4 из
sklearn.model_selection import train_test_split 5 из
pyimagesearch.preprocessing import AspectAwarePreprocessor 6 из pyimagesearch.io
import HDF5DatasetWriter 7 из imutils import paths 8 import numpy as np 9 импорт
индикатора выполнения 10 импорт json 11 импорт cv2 12 импорт ОС

```

Строки 2-12 импортируют необходимые пакеты Python. Мне нравится импортировать файл конфигурации нашего проекта в качестве первого импорта в проекте (строка 2). Этот метод — дело вкуса, поэтому не стесняйтесь размещать импорт в любом месте файла. Я также переименовываю Dogs_vs_cats_config в просто config, чтобы сделать его менее подробным при написании кода.

Оттуда остальные импорты, с которыми вы уже сталкивались в предыдущих главах; однако я хотел бы обратить ваше внимание на HDF5DatasetWriter в строке 6, тот самый HDF5DatasetWriter, который мы определили в главе 3. Этот класс будет использоваться для упаковки необработанных изображений на диске в один сериализованный файл.

Мы также снова воспользуемся модулем индикатора выполнения, простой служебной библиотекой, которую мне нравится использовать для измерения приблизительного времени выполнения данной задачи. Этот модуль совершенно не имеет отношения к глубокому обучению, но опять же, я считаю его удобным в использовании, так как для больших наборов данных может потребоваться несколько часов, чтобы упаковать набор данных изображений в формат HDF5.

Затем давайте возьмем пути к изображениям в наборе данных Kaggle Dogs vs. Cats:

```

14 # получаем пути к изображениям 15
trainPaths = list(paths.list_images(config.IMAGES_PATH)) 16 trainLabels =
[p.split(os.path.sep)[2].split(".")[0] for p в пути поезда]

```

```
18 le = LabelEncoder() 19 trainLabels =
le.fit_transform(trainLabels)
```

Набор данных Dogs vs. Cats имеет следующую примерную структуру каталогов:

```
kaggle_dogs_vs_cats/train/cat.11866.jpg
...
kaggle_dogs_vs_cats/train/dog.11046.jpg
```

Обратите внимание, как имя класса встроено в фактическое имя файла. Поэтому нам нужно извлечь файловую составляющую пути к файлу, разделенную на . разделитель и извлечь имя класса — собственно, именно это и делают строки 16 и 17 .

Имея пути ко всем изображениям в наборе данных, они перебирают их по отдельности и извлекают метки из путей к файлам. Если вы находитите эти строки кода запутанными, я бы посоветовал потратить секунду, чтобы вручную поиграть с кодом, в частности с переменной os.path.sep и функцией .split , используемой в строке пути к файлу, чтобы увидеть, как используются эти утилиты. для управления путями к файлам.

Строки 18 и 19 затем кодируют метки классов. Для проекта Kaggle Dogs vs. Cats нам понадобится три сплита: тренировочный сплит, проверочный сплит и тестовый сплит.

Наш следующий блок кода обрабатывает создание каждого из этих разделений:

```
21 # выполнить стратифицированную выборку из обучающей выборки для
построения 22 # тестового разделения из обучающих данных 23 split =
train_test_split(trainPaths, trainLabels,
24         test_size=config.NUM_TEST_IMAGES, stratify=trainLabels, random_state=42)
25
26 (trainPaths, testPaths, trainLabels, testLabels) = разделить
27
28 # выполнить еще одну стратифицированную выборку, на этот раз для
построения 29 # данных проверки
30 split = train_test_split(trainPaths, trainLabels,
31         test_size=config.NUM_VAL_IMAGES, stratify=trainLabels, random_state=42)
32
33 (trainPaths, valPaths, trainLabels, valLabels) = разделить
```

В строках 23-26 мы берем наши входные изображения и метки и используем их для построения разделения обучения и тестирования. Однако нам нужно выполнить еще одно разделение в строках 30-33 , чтобы создать набор проверки . Проверочный набор (почти всегда) берется из обучающих данных. Размер разделений для тестирования и проверки контролируется с помощью NUM_TEST_IMAGES и NUM_VAL_IMAGES , каждый из которых определен в нашем файле конфигурации.

Теперь, когда у нас есть разделы для обучения, тестирования и проверки, давайте создадим простой список, который будет позволяют нам перебирать их и эффективно записывать изображения в каждом наборе данных в наш файл HDF5:

```
35 # создать список, объединяющий пути обучения, проверки и тестирования 36 # пути к изображениям вместе с
соответствующими метками и выходными файлами HDF5 37 #
38 наборов данных = [
39     ("train", trainPaths, trainLabels, config.TRAIN_HDF5), ("val", valPaths,
40     valLabels, config.VAL_HDF5), ("test", testPaths, testLabels,
41     config.TEST_HDF5)]
```

```

42
43 # инициализируем препроцессор изображений и списки каналов RGB
44 # средние
45 aap = AspectAwarePreprocessor(256, 256)
46 (R, G, B) = ([], [], [])

```

В строке 38 мы определяем список наборов данных , который включает наше обучение, проверку и тестирование . переменные. Каждая запись в списке представляет собой 4-кортеж, состоящий из:

1. Название разделения (например, обучение, тестирование или проверка).
2. Соответствующие пути изображения для разделения.
3. Ярлыки для сплита.
4. Путь к выходному файлу HDF5 для разделения.

Затем мы инициализируем наш AspectAwarePreprocessor в строке 45 , используемой для изменения размера изображения до 256× . 256 пикселей (с учетом соотношения сторон изображения) перед записью в HDF5. Хорошо также инициализируйте три списка в строке 46 — R, G и B, используемые для хранения средней интенсивности пикселей для каждого канала.

Наконец, мы готовы построить наши наборы данных HDF5:

```

48 # цикл по кортежам набора данных
49 для (dType, paths, labels, outputPath) в наборах данных:
50     # создаем средство записи HDF5
51     print("[INFO] здание {}".format(outputPath))
52     писатель = HDF5DatasetWriter((len(paths), 256, 256, 3), outputPath)
53
54     # инициализируем индикатор выполнения
55     widgets = ["Строительный набор данных: ", progressbar.Percentage(), " ",
56               progressbar.Bar(), " ", progressbar.ETA()]
57     pbar = progressbar.ProgressBar(maxval = len (пути),
58                                   виджеты=виджеты).start()

```

В строке 49 мы начинаем перебирать каждое из четырех значений кортежа в списке наборов данных . Для каждого разделение данных , мы создаем экземпляра HDF5DatasetWriter в строке 52. Здесь размеры вывода набор данных будет (len(paths), 256, 256, 3), подразумевая, что всего изображений len(paths) , каждый из них имеет ширину 256 пикселей, высоту 256 пикселей и 3 канала.

Строки 54-58 затем инициализируют наш индикатор выполнения, чтобы мы могли легко отслеживать процесс генерации набора данных. Опять же, этот блок кода (вместе с остальными вызовами функции индикатора выполнения) является совершенно необязательным, поэтому не стесняйтесь комментировать их, если хотите.

Затем давайте запишем каждое изображение в заданном разделении данных в модуль записи:

```

60     # цикл по путям к изображениям
61     для (i, (путь, метка)) в перечислении (zip (пути, метки)):
62         # загрузить изображение и обработать его
63         изображение = cv2.imread (путь)
64         изображение = aap.preprocess(изображение)
65
66         # если мы строим обучающий набор данных, то вычисляем
67         # среднее значение каждого канала в изображении, затем обновите
68         # соответствующие списки
69         если dType == "поезд":
70             (b, g, r) = cv2.mean(изображение)[:3]
71             R.добавить(r)

```

```

72             G. добавить (g)
73             Б. добавить (б)
74
75         # добавляем изображение и метку # в набор данных HDF5
76         write.add([image], [label]) pbar.update(i)
77
78
79     # закрыть модуль записи HDF5
80     pbar.finish()
81     писатель.close()

```

В строке 61 мы начинаем перебирать каждое отдельное изображение и соответствующую метку класса в разделении данных. Строки 63 и 64 загружают изображение с диска, а затем применяют наш препроцессор с учетом аспектов для изменения размера изображения до 256x 256 пикселей.

Мы проверяем строку 69 , чтобы увидеть, изучаем ли мы разделение данных о поездах , и, если да, мы вычисляем среднее значение красного, зеленого и синего каналов (строка 70) и обновляем их соответствующие списки в строках 71-73. Вычисление среднего значения каналов RGB выполняется только для тренировочного набора и является обязательным, если мы хотим применить нормализацию вычитания среднего.

Строка 76 добавляет соответствующее изображение и метку в наш HDF5DatasetWriter. Как только все изображения в разделении данных будут сериализованы в набор данных HDF5, мы закрываем средство записи в строке 81.

Последним шагом является сериализация наших средних значений RGB на диск:

```

83 # создать словарь средних значений, затем сериализовать средние значения в
84 # JSON-файл
85 print("[INFO] сериализация означает...")
86 D = {"R": np.mean(R), "G": np.mean(G), "B": np.mean(B)} 87 f =
87 open(config.DATASET_MEAN, "w") 88 f.write(json.dumps(D)) 89 f.close()

```

Строка 86 создает словарь Python для средних значений RGB по всем изображениям в обучающем наборе. Имейте в виду, что каждый отдельный R, G и B содержит среднее значение канала для каждого изображения в наборе данных. Вычисление среднего значения этого списка дает нам среднее значение интенсивности пикселей для всех изображений в списке. Затем этот словарь сериализуется на диск в формате JSON в строках 87-88.

Давайте продолжим и сериализуем набор данных Kaggle Dogs vs. Cats в формат HDF5. Откройте терминал, а затем введите следующую команду:

```

$ python build_dogs_vs_cats.py [INFO]
сборка kaggle_dogs_vs_cats/hdf5/train.hdf5...
Набор данных здания: 100% ##### | Время: 0:02:39 [ИНФО]
сборка kaggle_dogs_vs_cats/hdf5/val.hdf5...
Набор данных здания: 100% ##### | Время: 0:00:20 [INFO]
сборка kaggle_dogs_vs_cats/hdf5/test.hdf5...
Набор данных здания: 100% ##### | Время: 0:00:19

```

Как вы можете видеть из моего вывода, файл HDF5 был создан для каждого из разделов обучения, тестирования и проверки. Генерация тренировочного разделения заняла больше всего времени, так как это разделение содержало больше всего данных (2 мин 39 с). Сплиты для тестирования и проверки заняли значительно меньше времени (20 с) из -за того, что в этих сплитах меньше данных.

Мы можем увидеть каждый из этих выходных файлов на нашем диске, перечислив содержимое каталога hdf5:

```
$ ls -l ../datasets/kaggle_dogs_vs_cats/hdf5/ всего 38400220
-rw-rw-r-- 1 адриан адриан 3932182144 7 апр 18:00 test.hdf5 -rw-rw-r-- 1 адриан
адриан 31457442144 7 апр 17:59 train.hdf5 -rw-rw-r-- 1 адриан Адриан
3932182144 7 апр 18:00 val.hdf5
```

Глядя на эти размеры файлов, вы можете быть немного удивлены. Необработанные изображения Kaggle Dogs vs. Cats , находящиеся на диске, имеют размер всего 595 МБ — почему файлы .hdf5 такие большие? Только файл train.hdf5 весит 31,45 ГБ, а файлы test.hdf5 и val.hdf5 — почти 4 ГБ. Почему?

Имейте в виду, что форматы необработанных файлов изображений, такие как JPEG и PNG, применяют алгоритмы сжатия данных, чтобы сохранить небольшой размер файла изображения. Однако мы эффективно убрали все типы сжатия и сохраняем изображения в виде необработанных массивов NumPy (т. е. растровых изображений). Это отсутствие сжатия резко увеличивает наши затраты на хранение, но также поможет ускорить время обучения, поскольку нам не придется тратить время процессора на декодирование изображения — вместо этого мы можем получить доступ к изображению непосредственно из набора данных HDF5, предварительно обработать его и передать. это через нашу сеть.

Давайте также взглянем на наш средний файл RGB:

```
$ cat output/dogs_vs_cats_mean.json {"B": 106.13178224639893, "R": 124.96761639328003, "G": 115.97504255599975}
```

Здесь мы видим, что красный канал имеет среднюю интенсивность пикселей 124,96 для всех изображений в наборе данных. Синий канал имеет среднее значение 106,13, а зеленый канал — среднее значение 115,97. Мы создадим новый препроцессор изображений, чтобы нормализовать наши изображения, вычитая эти средние значения RGB из входных изображений перед их передачей через нашу сеть. Эта нормализация среднего помогает «центрировать» данные вокруг нулевого среднего. Как правило, эта нормализация позволяет нашей сети обучаться быстрее, и именно поэтому мы используем этот тип нормализации (а не масштабирование [0,1] для больших и сложных наборов данных).

9.4 Резюме

В этой главе мы узнали, как сериализовать необработанные изображения в набор данных HDF5, подходящий для обучения глубокой нейронной сети. Причина, по которой мы сериализовали необработанные изображения в файл HDF5, а не просто обращались к мини-пакетам путей к изображениям на диске при обучении, связана с задержкой ввода-вывода — для каждого изображения на диске нам пришлось бы выполнять операцию ввода-вывода для чтения. Изображение. Эта тонкая оптимизация не кажется большой проблемой, но задержка ввода-вывода является огромной проблемой в конвейере глубокого обучения — процесс обучения уже достаточно медленный, и если мы усложним доступ к нашим данным для наших сетей, мы только дальше стреляем себе в ногу.

И наоборот, если мы сериализуем все изображения в эффективно упакованный файл HDF5, мы можем использовать очень быстрые фрагменты массива для извлечения наших мини-пакетов, тем самым значительно уменьшая задержку ввода-вывода и помогая ускорить процесс обучения. Всякий раз, когда вы используете библиотеку Keras и работаете с набором данных, слишком большим, чтобы поместиться в память, обязательно сначала рассмотрите возможность сериализации набора данных в формате HDF5 — как мы узнаем в следующей главе, это упрощает обучение вашей сети (и более эффективная) задача.

10. Соревнования в Kaggle: Dogs vs. Cats

В нашей предыдущей главе мы узнали, как работать с HDF5 и наборами данных, которые слишком велики, чтобы поместиться в память. Для этого мы определили служебный скрипт Python, который можно использовать для получения входного набора данных изображений и их сериализации в высокоеффективный набор данных HDF5. Представление набора изображений в наборе данных HDF5 позволяет нам избежать проблем с задержкой ввода-вывода, тем самым ускоряя процесс обучения.

Например, если бы мы определили генератор набора данных, который последовательно загружал изображения с диска, нам потребовалось бы N операций чтения, по одной для каждого изображения. Однако, поместив наш набор данных изображений в набор данных HDF5, мы можем вместо этого загружать пакеты изображений, используя одно чтение. Это действие значительно сокращает количество вызовов ввода-вывода и позволяет нам работать с очень большими наборами данных изображений.

В этой главе мы собираемся расширить нашу работу и узнать, как определить генератор изображений для наборов данных HDF5, подходящий для обучения сверточных нейронных сетей с помощью Keras. Этот генератор откроет набор данных HDF5, выдаст пакеты изображений и связанных обучающих меток для обучения сети, и продолжит делать это до тех пор, пока наша модель не достигнет достаточно низких потерь/высокой точности.

Чтобы выполнить этот процесс, мы сначала рассмотрим три новых препроцессора изображений, предназначенных для повышения точности классификации: вычитание среднего, извлечение фрагментов и кадрирование (также называемое 10-кадрированием или передискретизацией). После того, как мы определили наш новый набор препроцессоров, мы перейдем к определению фактического генератора набора данных HDF5.

Оттуда мы реализуем основополагающую архитектуру AlexNet из статьи Крижевского и др. 2012 года «Классификация ImageNet с глубокими сверточными нейронными сетями» [6]. Затем эта реализация AlexNet будет обучена решению задачи Kaggle Dogs vs. Cats. Учитывая обученную модель, мы оценим ее производительность на тестовом наборе, а затем воспользуемся методами избыточной выборки для дальнейшего повышения точности классификации. Как покажут наши результаты, наша сетевая архитектура + методы обрезки позволят нам занять место в топ-25 лидеров конкурса Kaggle Dogs vs. Cats .

10.1 Дополнительные препроцессоры изображений

В этом разделе мы реализуем два новых препроцессора изображений: 1. Препроцессор вычитания среднего, предназначенный для вычитания средней интенсивности красных, зеленых и синих пикселей в наборе данных из входного изображения (что является формой нормализации данных).).

2. Препроцессор патчей, используемый для случайного извлечения областей $M \times N$ пикселей из изображения во время обучения.
3. Препроцессор передискретизации, используемый во время тестирования для выборки пяти областей входного изображения. (четыре угла + центральная область) вместе с соответствующими им горизонтальными переворотами (всего 10 культур).

Используя избыточную выборку, мы можем повысить точность нашей классификации, пропустив 10 культур через наш CNN, а затем усреднение по 10 прогнозам.

10.1.1 Средняя предварительная обработка

Начнем со среднего препроцессора. В главе 9 мы узнали, как преобразовать изображение набор данных в формат HDF5 — часть этого преобразования включала вычисление среднего красного, зеленого и Интенсивность синих пикселей на всех изображениях во всем наборе данных. Теперь, когда у нас есть эти средние значения, мы собираемся выполнить попиксельное вычитание этих значений из наших входных изображений в виде формы нормализации данных. Учитывая входное изображение I и его каналы R, G, B, мы можем выполнить среднее вычитание через:

- $R = R - \mu R$
- $G = G - \mu G$
- $B = B - \mu B$

Где μR , μG и μB вычисляются при преобразовании набора данных изображения в формат HDF5.

Рисунок 10.1 включает визуализацию вычитания средних значений RGB из входного изображения.

обратите внимание, как вычитание выполняется попиксельно.



$$\begin{aligned} R &= 124.96 \\ - G &= 115.97 = \\ B &= 106.13 \end{aligned}$$

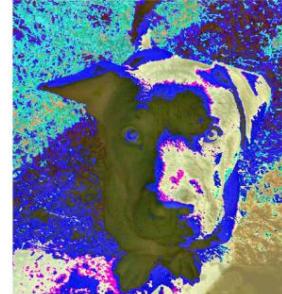


Рисунок 10.1: Пример применения вычитания среднего к входному изображению (слева) путем вычитания $R = 124,96$, $G = 115,97$, $B = 106,13$ по пикселям, что дает выходное изображение (справа). Иметь в виду вычитание используется для уменьшения влияния изменений освещения во время классификации.

Чтобы сделать эту концепцию более конкретной, давайте продолжим и реализуем наш MeanPreprocessor. класс:

```
--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- nn
| |--- предварительная обработка
| | |--- __init__.py
| | |--- аспектаварепропроцессор.py
| | |--- imagedoarraypreprocessor.py
| | |--- означаетпропроцессор.py
| | |--- simplepreprocessor.py
| |--- утилиты
```

Обратите внимание, как я поместил новый файл с именем meanpreprocessor.py в препроцессор . подмодуль pyimagesearch — в этом месте будет жить наш класс MeanPreprocessor .
Давайте продолжим и реализуем этот класс сейчас:

```

1 # импортируем необходимые пакеты
2 import cv2
3
4 класс MeanPreprocessor:
5     def __init__(self, rMean, gMean, bMean):
6         # сохранить средние значения красного, зеленого и синего каналов в
7         # Обучающий набор
8         self.rMean = rMean
9         self.gMean = gMean
10        self.bMean = bMean

```

Строка 5 определяет конструктор MeanPreprocessor, который требует три аргумента:
соответствующие красные, зеленые и синие средние значения, вычисленные по всему набору данных. Эти значения
затем сохраняется в строках 8-10.

Далее давайте определим метод препроцессора , обязательную функцию для каждого препроцессора, который мы
намерен применить к нашему конвейеру обработки изображений:

```

12     предварительная обработка def (я, изображение):
13         # разделить изображение на соответствующий красный, зеленый и синий
14         # каналы
15         (B, G, R) = cv2.split(image.astype("float32"))
16
17         # вычесть средства для каждого канала
18         R -= self.rMean
19         G -= self.gMean
20         B -= self.bMean
21
22         # объединить каналы обратно вместе и вернуть изображение
23         вернуть cv2.merge ([B, G, R])

```

В строке 15 используется функция cv2.split для разделения входного изображения на соответствующие компоненты RGB.
Имейте в виду, что OpenCV представляет изображения в порядке BGR, а не в формате RGB ([38],
<http://pyimg.co/rrao>), поэтому наш возвращаемый кортеж имеет сигнатуру (B, G, R) , а не (R, G,
Б). Мы также обеспечим, чтобы эти каналы имели тип данных с плавающей запятой, поскольку изображения OpenCV
обычно представляются как 8-битные целые числа без знака (в этом случае у нас не может быть отрицательных значений, и
вместо этого будет выполняться арифметика по модулю).

Строки 17-20 выполняют само вычитание среднего, вычитая соответствующие средние значения RGB.
из каналов RGB входного изображения. Стока 23 затем объединяет нормализованные каналы обратно.
вместе и возвращает полученное изображение вызывающей функции.

10.1.2 Предварительная обработка патчей

PatchPreprocessor отвечает за случайную выборку $M \times N$ областей изображения во время
тренировочный процесс. Мы применяем предварительную обработку исправлений, когда пространственные размеры наших входных изображений
больше, чем ожидает CNN - это распространенный метод, помогающий уменьшить переоснащение,
и, следовательно, является формой регуляризации. Вместо того, чтобы использовать все изображение во время обучения, мы
вместо этого обрезать случайную часть и передать в сеть (см. рис. 10.2, где показан пример
предварительная обработка урожая).

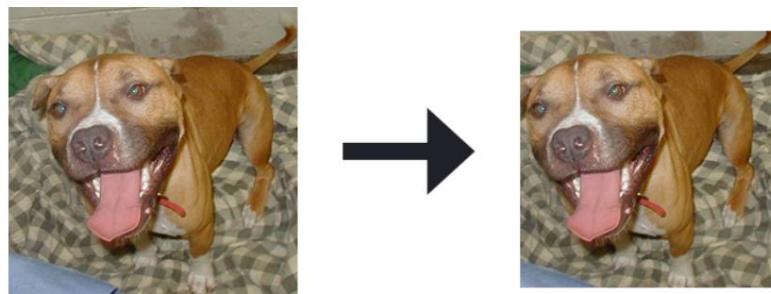


Рисунок 10.2: Слева: исходное исходное изображение 256× 256 . Справа: случайное кадрирование 227× 227 область с изображения.

Применение этой обрезки означает, что сеть никогда не увидит одно и то же изображение (если только случайное стечание обстоятельств), аналогично дополнению данных. Как вы знаете из предыдущей главы, мы построил набор данных HDF5 изображений Kaggle Dogs vs. Cats, где каждое изображение имеет размер 256 × 256 пикселей. Однако архитектура AlexNet, которую мы реализуем позже в этой главе, может принимать только изображения размером 227× 227 пикселей.

Итак, что нам делать? Примените `SimplePreprocessor`, чтобы изменить размер каждого из 256× 256 пикселей до 227 × 227? Нет, это было бы расточительно, тем более, что это отличный возможность выполнять аугментацию данных путем случайного вырезания области 227 × 227 из изображение 256× 256 во время тренировки — собственно, именно так проходит этот процесс Крижевский и др. поезда AlexNet в наборе данных ImageNet.

`PatchPreprocessor`, как и все другие препроцессоры изображений, будет сортироваться в подмодуль предварительной обработки `pyimagesearch`:

```
--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- нн
| |--- предварительная обработка
| | |--- __init__.py
| | |--- аспектаварепропроцессор.py
| | |--- imagedtoarrayprocessor.py
| | |--- означает препроцессор.py
| | |--- patchprocessor.py
| | |--- simpleprocessor.py
| |--- утилиты
```

Откройте файл `patchprocessor.py` и давайте определим класс `PatchProcessor`:

```
1 # импортируем необходимые пакеты
2 из sklearn.feature_extraction.image импорта extract_patches_2d
3
4 класс PatchProcessor:
5     def __init__(я, ширина, высота):
6         # сохранить целевую ширину и высоту изображения
7         собственная ширина = ширину
8         self.height = высота
```

Строка 5 определяет конструкцию PatchPreprocessor — нам просто нужно указать цель ширина и высота обрезанного изображения.

Затем мы можем определить функцию предварительной обработки:

```
10     предварительная обработка def (я, изображение):
11         # извлекаем случайную обрезку изображения с заданной шириной
12         # и высота
13         вернуть extract_patches_2d (изображение, (self.height, self.width),
14             max_patches=1)[0]
```

Извлечь случайные патчи размером `self.width x self.height` легко с помощью `extract_patches_2d` функция из библиотеки scikit-learn. Учитывая входное изображение, эта функция случайным образом извлекает патч из образа. Здесь мы указываем `max_patches=1`, указывая, что нам нужен только один случайный патч из входного изображения.

Класс `PatchPreprocessor` не кажется чем-то большим, но на самом деле он очень эффективен. метод, чтобы избежать переобучения, применяя еще один уровень увеличения данных. Мы будем использовать `PatchPreprocessor` при обучении AlexNet. Следующий препроцессор, `CropPreprocessor`, будет использоваться при оценке нашей обученной сети.

10.1.3 Предварительная обработка урожая

Затем нам нужно определить препроцессор `CropPreprocessor`, ответственный за вычисление 10-кратного урожая для избыточной выборки. На этапе оценки нашей CNN мы обрежем четыре угла входного изображения + центральную область, а затем выполнить соответствующие им горизонтальные перевороты, всего десять сэмплов на входное изображение (рис. 10.3).

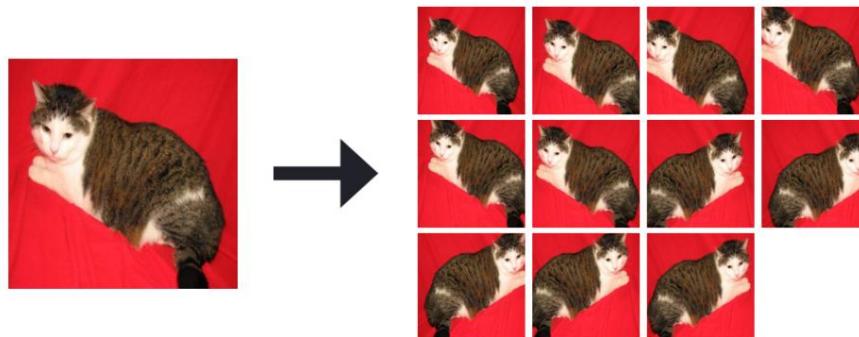


Рисунок 10.3: Слева: Исходное входное изображение 256×256 . Справа: применение препроцессора 10-crop для извлечения десяти кадров изображения 227×227 , включая центр, четыре угла и их края. соответствующие горизонтальные зеркала.

Эти десять выборок будут пропущены через CNN, а затем вероятности усреднены. Применение этого метода избыточной выборки, как правило, увеличивает точность классификации на 1-2 процента. (а в некоторых случаях и выше).

Класс `CropPreprocessor` также будет находиться в подмодуле предварительной обработки `pyimagesearch`:

```
-- pyimagesearch
| |-- __init__.py
| |-- обратные вызовы
| |-- нн
```

```

|--- предварительная обработка
| |--- __init__.py
| |--- аспектаварепропроцессор.py
| |--- урожаяпропропроцессор.py
| |--- imagedtoarrayprocessor.py
| |--- означает пропропроцессор.py
| |--- patchprocessor.py
| |--- simpleprocessor.py
|--- утилиты

```

Откройте файл cropprocessor.py и давайте определим его:

```

1 # импортируем необходимые пакеты
2 импортировать numpy как np
3 импорт cv2
4
5 класс CropPreprocessor:
6     def __init__(self, width, height, horiz=True, inter=cv2.INTER_AREA):
7         # сохранить ширину, высоту целевого изображения, независимо от того,
8         # горизонтальные сальто должны быть включены вместе с
9         # метод интерполяции, используемый при изменении размера
10        собственная ширина = ширина
11
12        self.height = высота
13        self.horiz = горизонт
14
15        self.inter = интер

```

Строка 6 определяет конструктор `CropPreprocessor`. Единственными обязательными аргументами являются целевая ширина и высота каждой обрезанной области. Мы также можем дополнительно указать, необходимо применить горизонтальное отражение (по умолчанию `True`) вместе с алгоритмом интерполяции `OpenCV` будет использовать для изменения размера. Все эти аргументы хранятся внутри класса для использования внутри метода предварительной обработки.

Говоря об этом, давайте теперь определим метод предварительной обработки:

```

15    предварительная обработка def (я, изображение):
16        # инициализируем список культур
17        урожай = []
18
19        # получить ширину и высоту изображения, а затем использовать их
20        # размеры для определения углов изображения на основе
21        (ч, ш) = изображение.форма[:2]
22        координаты = [
23            [0, 0, собственная ширина, собственная высота],
24            [w - собственная ширина, 0, w, собственная высота],
25            [w - собственная ширина, h - собственная высота, w, h],
26            [0, h - self.height, self.width, h]]
27
28        # также вычисляем центр кадрирования изображения
29        dW = int (0,5 * (w - собственная ширина))
30        dH = int (0,5 * (h - собственная высота))
31        coords.append([dW, dH, w - dW, h - dH])

```

Метод предварительной обработки требует только одного аргумента — изображения , которое мы собираемся применять передискретизацию. Мы получаем ширину и высоту входного изображения в строке 21, которые затем

позволяет нам вычислить (x, y)-координаты четырех углов (верхний левый, верхний правый, нижний правый, внизу слева соответственно) в строках 22-26. Центральная обрезка изображения затем вычисляется по линиям . 29 и 30, затем добавлены в список координат в строке 31.

Теперь мы готовы извлечь каждую из культур:

```

33     # перебираем координаты, извлекаем каждую культуру,
34     # и изменить размер каждого из них до фиксированного размера
35     for (startX, startY, endX, endY) в координатах:
36         кадрирование = изображение[началоШ:конецY, началоЧ:конецХ]
37         обрезка = cv2.resize (обрезка, (собственная ширина, собственная высота),
38                             интерполяция = self.inter)
39         урожай.append (обрезка)

```

В строке 35 мы перебираем каждую начальную и конечную (x, y)-координаты прямоугольного урожай. Стока 36 извлекает урожай с помощью нарезки массива NumPy, размер которого мы затем изменяем в строке 37 до убедитесь, что целевые размеры ширины и высоты соблюdenы. Культура добавляется в список культур.

В случае, когда необходимо вычислить горизонтальные зеркала, мы можем перевернуть каждое из пяти исходных культур, оставив нам всего десять культур:

```

41     # проверяем, нужно ли делать горизонтальные сальто
42     если self.goriz.:
43         # вычисляем горизонтальное зеркальное отражение для каждой культуры
44         зеркала = [cv2.flip(c, 1) для c в посевах]
45         урожай.продлить(зеркала)
46
47     # вернуть набор урожая
48     вернуть np.array (урожай)

```

Затем массив культур возвращается в вызывающую функцию в строке 48. Используя как MeanPreprocessor для нормализации и CropPreprocessor для передискретизации . в состоянии получить более высокую точность классификации, чем это возможно в противном случае.

10.2 Генераторы наборов данных HDF5

Прежде чем мы сможем реализовать архитектуру AlexNet и обучить ее набору данных Kaggle Dogs vs. Cats, сначала нам нужно определить класс, отвечающий за получение пакетов изображений и меток из нашего HDF5. набор данных. В главе 9 обсуждалось, как преобразовать набор изображений, находящихся на диске, в набор данных HDF5. но как нам вернуть их обратно?

Ответ заключается в определении класса HDF5DatasetGenerator в подмодуле io.pyimagesearch :

```

--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ио
| | |--- __init__.py
| | |--- hdf5datasetgenerator.py
| | |--- hdf5datasetwriter.py
| |--- ии
| |--- предварительная обработка
| |--- утилиты

```

Раньше все наши наборы данных изображений можно было загружать в память, чтобы мы могли положиться на Keras. утилиты-генераторы для получения наших пакетов изображений и соответствующих меток. Однако теперь, когда наш наборы данных слишком велики, чтобы поместиться в память, нам нужно самим реализовать этот генератор.

Откройте файл hdf5datasetgenerator.py, и мы приступим к работе:

```

1 # импортируем необходимые пакеты
2 из keras.utils импортировать np_utils
3 импортировать numpy как np
4 импортировать h5py
5
6 класс HDF5DatasetGenerator:
7     def __init__(self, dbPath, batchSize, препроцессоры = нет,
8                  aug=Нет, бинаризация=Истина, классы=2):
9         # сохранить размер пакета, препроцессоры и модуль расширения данных,
10        # следует ли бинаризировать метки вместе с
11        # общее количество классов
12        self.batchSize = размер партии
13        self.preprocessors = препроцессоры
14        self.aug = август
15        self.binarize = бинаризировать
16        self.classes = классы
17
18        # открываем базу данных HDF5 на чтение и определяем общее количество
19        # количество записей в базе
20        self.db = h5py.File(dbPath)
21        self.numImages = self.db["метки"].shape[0]

```

В строке 7 мы определяем конструктор для нашего HDF5DatasetGenerator. Этот класс принимает количество аргументов, два из которых обязательны, а остальные необязательны. Я подробно описал каждую аргументы ниже:

- dbPath: путь к нашему набору данных HDF5, в котором хранятся наши изображения и соответствующие метки классов.
- batchSize: размер мини-пакетов, которые будут получены при обучении нашей сети.
- препроцессоры: список препроцессоров изображений, которые мы собираемся применить (например, MeanPreprocessor, Препроцессор ImageToArray и др.).
- aug: по умолчанию None, мы также можем предоставить Keras ImageDataGenerator для применения данных .расширение непосредственно внутри нашего HDF5DatasetGenerator.
- бинаризация: обычно мы будем хранить метки классов как отдельные целые числа внутри нашего набора данных HDF5; однако, как мы знаем, если мы применяем категориальную кросс-энтропию или бинарную кросс-энтропию как нашей функции потерь, нам сначала нужно бинаризировать метки как векторы с горячим кодированием — этот переключатель указывает, должна ли выполняться эта бинаризация (по умолчанию она равна True).
- классы: количество уникальных меток классов в нашем наборе данных. Это значение необходимо для точно построить наши горячие закодированные векторы на этапе бинаризации.

Эти переменные хранятся в строках 12-16, поэтому мы можем получить к ним доступ из остальной части класса.

Строка 20 открывает указатель файла на наш файл набора данных HDF5. Стока 21 создает вспомогательную переменную, используемую для доступа к общему количеству точек данных в наборе данных.

Далее нам нужно определить функцию-генератор , которая, как следует из названия, отвечает за предоставление пакетов изображений и меток классов функции Keras .fit_generator при обучении сеть:

```

23     генератор деф (я, проходит = np.inf):
24         # инициализируем счетчик эпох

```

```

25     эпохи = 0
26
27     # продолжать цикл бесконечно -- модель остановится, как только мы получим
28     # достичь нужного количества эпох
29     в то время как эпохи <проходят:
30         # цикл по набору данных HDF5
31         для i в np.arange(0, self.numImages, self.batchSize):
32             # извлечь изображения и метки из набора данных HDF
33             изображения = self.db["images"][:i: i + self.batchSize]
34             labels = self.db["labels"][:i: i + self.batchSize]

```

Строка 23 определяет функцию генератора , которая может принимать необязательный аргумент pass. Думайте о количестве проходов как об общем количестве эпох — в большинстве случаев мы не хотим, чтобы наши генератор должен быть связан с общим количеством эпох; наша методика обучения (фиксированное количество эпох, ранняя остановка и т. д.) должны отвечать за это. Однако в определенных ситуациях это часто полезно предоставить эту информацию генератору.

В строке 29 мы начинаем цикл по количеству желаемых эпох — по умолчанию этот цикл будет работать на неопределенный срок до:

1. Крас достигает критерия завершения обучения.
2. Явно останавливаем процесс обучения (т.е. ctrl+c).

Строка 31 начинает цикл по каждому пакету точек данных в наборе данных. Мы извлекаем изображения и метки размера batchSize из нашего набора данных HDF5 в строках 33 и 34.

Затем давайте проверим, должны ли метки быть закодированы горячим способом:

```

36     # проверяем, должны ли метки быть бинарными
37     если self.binarize:
38         метки = np_utils.to_categorical(метки,
39                                         self.классы)

```

Затем мы также можем увидеть, следует ли применять какие-либо препроцессоры изображений:

```

41     # проверяем, не являются ли наши препроцессоры None
42     если self.preprocessors не None:
43         # инициализируем список обработанных изображений
44         procImages = []
45
46         # цикл по изображениям
47         для изображения в изображениях:
48             # цикл по препроцессорам и применение каждого
49             # к изображению
50             для p в self.preprocessors:
51                 изображение = p.preprocess(изображение)
52
53             # обновить список обработанных изображений
54             procImages.append(изображение)
55
56             # обновить массив изображений для обработки
57             # картинки
58             изображения = np.array(procImages)

```

При условии, что препроцессоры не равны None (строка 42), мы перебираем каждое изображение в цикле. Пакет и примените каждый из препроцессоров , вызывав метод препроцессора на отдельном изображении. Это позволяет нам объединить несколько препроцессоров изображений.

Например, наш первый препроцессор может изменить размер изображения до фиксированного размера с помощью нашего SimplePreprocessor. класс. Оттуда мы можем выполнить среднее вычитание через MeanPreprocessor. И после этого, нам нужно будет преобразовать изображение в массив, совместимый с Keras, с помощью ImageToArrayPreprocessor. На этом этапе должно быть ясно, почему мы определили все наши классы предварительной обработки с помощью препроцессора . метод — он позволяет нам объединить наши препроцессоры внутри генератора данных. Затем предварительно обработанные изображения преобразуются обратно в массив NumPy в строке 58.

При условии, что мы предоставили экземпляр aug, класс ImageDataGenerator , используемый для данных aug . Мы также хотим применить увеличение данных к изображениям:

```
60             # если аугментатор данных существует, применяем его
61         если self.aug не None:
62             (изображения, метки) = следующий (self.aug.flow (изображения,
63                                         метки, batch_size=self.batchSize))
```

Наконец, мы можем передать вызывающему генератору Keras 2 пакета изображений и меток :

```
65             # получить кортеж изображений и меток
66         выход (изображения, ярлыки)
67
68             # увеличить общее количество эпох
69             эпохи += 1
70
71     защита близко (я):
72         # закрыть базу данных
73         self.db.close()
```

Строка 69 увеличивает наше общее количество эпох после того, как все мини-пакеты в наборе данных были обработанный. Метод close в строках 71-73 просто отвечает за закрытие указателя на Набор данных HDF5.

По общему признанию, внедрение HDF5DatasetGenerator может не «ощущаться», как будто мы делаем что-либо глубокое обучение. В конце концов, разве это не просто класс, отвечающий за получение пакетов данных из файла? Технически да, это правильно. Однако имейте в виду, что практическое глубокое обучение — это больше, чем просто определение архитектуры модели, инициализация оптимизатора и применение его к набору данных.

На самом деле нам нужны дополнительные инструменты, облегчающие нам работу с наборами данных, особенно с наборами данных, которые слишком велики, чтобы поместиться в памяти. Как мы увидим в остальной части этого книги, наш HDF5DatasetGenerator пригодится несколько раз — и когда вы начнете создавая свои собственные приложения/эксперименты для глубокого обучения, вам очень повезет, если вы репертуар.

10.3 Внедрение AlexNet

Теперь давайте перейдем к реализации оригинальной архитектуры AlexNet Крижевского и др. Стол обобщая архитектуру AlexNet, можно увидеть в таблице 10.1.

Обратите внимание, что наши входные изображения предполагаются размером $227 \times 227 \times 3$ пикселя — на самом деле это правильный размер ввода для AlexNet. Как упоминалось в главе 9, в оригинальной публикации Крижевский и другие. сообщил, что входные пространственные размеры составляют $224 \times 224 \times 3$; однако, поскольку мы знаем 224×224 не возможно тайлингом с ядром 11×1 , мы предполагаем, что в публикации, вероятно, была опечатка, а 224×224 на самом деле должно быть 227×227 .

Первый блок AlexNet применяет 96 ядер 11×11 с шагом 4×4 , за которым следует Активация RELU и максимальное объединение с размером пула 3×3 и шагом 2×2 , что приводит к выходной объем размером 55×55 .

Тип слоя	Выходной размер	Размер фильтра/шаг
ВХОДНОЕ ИЗОБРАЖЕНИЕ 227× 227× 3		
КОНВ.	57× 57× 96 11× 11/4× 4, K = 96	
ДЕЙСТВОВАТЬ	57× 57× 96	
БН	57× 57× 96	
БАССЕЙН	16× 16× 96 3× 3/2× 2	
ВЫБЫВАТЬ	28× 28× 96	
КОНВ.	28× 28× 256 5× 5, K = 256	
ДЕЙСТВОВАТЬ	28× 28× 256	
БН	28× 28× 256	
БАССЕЙН	13× 13× 256 3× 3/2× 2	
ВЫБЫВАТЬ	13× 13× 256	
КОНВ.	13× 13× 384 3× 3, K = 384	
ДЕЙСТВОВАТЬ	13× 13× 384	
БН	13× 13× 384	
КОНВ.	13× 13× 384 3× 3, K = 384	
ДЕЙСТВОВАТЬ	13× 13× 384	
БН	13× 13× 384	
КОНВ.	13× 13× 256 3× 3, K = 256	
ДЕЙСТВОВАТЬ	13× 13× 256	
БН	13× 13× 256	
БАССЕЙН	13× 13× 256 3× 3/2× 2	
ВЫБЫВАТЬ	6× 6× 256	
ФК	4096	
ДЕЙСТВОВАТЬ	4096	
БН	4096	
ВЫБЫВАТЬ	4096	
ФК	4096	
ДЕЙСТВОВАТЬ	4096	
БН	4096	
ВЫБЫВАТЬ	4096	
ФК	1000	
СОФТМАКС	1000	

Таблица 10.1: Сводная таблица архитектуры AlexNet. Размеры выходного тома включены для каждого слоя вместе с размером сверточных фильтров/размером пула, если это необходимо.

Затем мы применяем второй слой CONV => RELU => POOL , на этот раз с использованием фильтров 256, 5×5 . с шагом 1×1 . После повторного применения максимального объединения с размером объединения 3×3 и шагом 2×2 мы остаются с объемом 13×13 .

Далее применяем (CONV=>RELU)*3=>POOL. Первые два слоя CONV изучают 384, 3×3 фильтра, в то время как окончательный CONV изучает 256 фильтров 3×3 .

После еще одной операции максимального объединения мы достигаем двух наших уровней FC , каждый из которых имеет 4096 узлов и Активации RELU между ними. Последний слой в сети — это наш классификатор softmax.

Когда AlexNet был впервые представлен, у нас не было таких методов, как нормализация пакетов. наша реализация, мы собираемся включить пакетную нормализацию после активации, как стандартно для большинства задач классификации изображений с использованием сверточных нейронных сетей. Также хорошо включать очень небольшое количество отсева после каждой операции POOL , чтобы еще больше уменьшить переоснащение.

Чтобы реализовать AlexNet, давайте создадим новый файл с именем alexnet.py в подмодуле сопр модуля nn в pyimagesearch:

```
--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ио
| |--- нн
| | |--- __init__.py
| | |--- конв.
| | | |--- __init__.py
| | |--- alexnet.py
...
|--- предварительная обработка
|--- утилиты
```

Оттуда откройте alexnet.py, и мы реализуем эту оригинальную архитектуру:

```
1 # импортируем необходимые пакеты
2 из импорта keras.models Последовательный
3 из keras.layers.normalization import BatchNormalization
4 из keras.layers.convolutional импорт Conv2D
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 импорта keras.regularizers |2
11 из keras импортировать бэкэнд как К
```

Строки 2-11 импортируют наши необходимые классы Keras — мы уже использовали все эти слои в предыдущих главах этой книги, поэтому я пропущу подробное описание каждого из них. Единственный импорт, который я хочу обратить ваше внимание на строку 10 , где мы импортируем функцию |2 — этот метод будет отвечает за применение снижения веса L2 к слоям веса в сети.

Теперь, когда мы позаботились об импорте, давайте начнем определение AlexNet:

```
13 класс AlexNet:
14     @статический метод
15     сборка def (ширина, высота, глубина, классы, reg = 0,0002):
16         # инициализируем модель вместе с входной формой, чтобы она была
```

```

17     # "каналы последние" и само измерение каналов
18     модель = Последовательный()
19     inputShape = (высота, ширина, глубина)
20     ЧанДим = -1
21
22     # если мы используем "сначала каналы", обновите форму ввода
23     # и размер каналов
24     если K.image_data_format() == "channels_first":
25         inputShape = (глубина, высота, ширина)
26         ЧанДим = 1

```

Строка 15 определяет метод сборки AlexNet. Как и во всех предыдущих примерах в этой книге, метод сборки необходим для построения фактической сетевой архитектуры и возврата ее в вызывающая функция. Этот метод принимает четыре аргумента: ширину, высоту и глубину .

входные изображения, за которыми следует общее количество меток классов в наборе данных. Необязательный параметр, reg, контролирует степень регуляризации L2, которую мы будем применять к сети. Для большего, более глубокого сетях, применение регуляризации имеет решающее значение для уменьшения переобучения при одновременном повышении точности наборы для проверки и тестирования.

Строка 18 инициализирует саму модель вместе с inputShape и измерением канала, предполагая , что мы используем порядок «каналы в последнюю очередь». Если вместо этого мы используем порядок «каналы в первую очередь», мы обновить inputShape и chanDim (строки 24-26).

Давайте теперь определим первый набор слоев CONV => RELU => POOL в сети:

```

28     # Блок №1: первый набор слоев CONV => RELU => POOL
29     model.add(Conv2D(96, (11, 11), шаги=(4, 4),
30                     input_shape = inputShape, padding = "такой же",
31                     kernel_regularizer=l2(reg)))
32     model.add(Активация("relu"))
33     model.add (Пакетная нормализация (ось = chanDim))
34     model.add(MaxPooling2D(pool_size=(3, 3), шаги=(2, 2)))
35     model.add (Выпадение (0,25))

```

Наш первый слой CONV изучит 96 фильтров, каждый размером 11×11 (строки 28 и 29), используя шаг. 4×4 . Применив параметр kernel_regularizer к классу Conv2D , мы можем применить наш Параметр регуляризации веса L2 — эта регуляризация будет применяться ко всем слоям CONV и FC . в сети.

Активация ReLU применяется после нашего CONV, за которым следует BatchNormalization (строки 32) . и 33). Затем применяется MaxPooling2D для уменьшения наших пространственных измерений (строка 34). Хорошо также применяйте отсев с небольшой вероятностью (25 процентов), чтобы уменьшить переоснащение (строки 35).

Следующий блок кода определяет другой набор слоев CONV => RELU => POOL , на этот раз обучая 256 фильтров размером 5×5 каждый:

```

37     # Блок №2: второй набор слоев CONV => RELU => POOL
38     model.add(Conv2D(256, (5, 5), padding="то же самое",
39                      kernel_regularizer=l2(reg)))
40     model.add(Активация("relu"))
41     model.add (Пакетная нормализация (ось = chanDim))
42     model.add(MaxPooling2D(pool_size=(3, 3), шаги=(2, 2)))
43     model.add (Выпадение (0,25))

```

Более глубокие и богатые функции изучаются в третьем блоке AlexNet, где мы складываем несколько CONV . => RELU вместе перед применением операции POOL:

```

45         # Блок №3: CONV => RELU => CONV => RELU => CONV => RELU
46         model.add(Conv2D(384, (3, 3), padding="то же самое",
47                           kernel_regularizer=l2(reg)))
48         model.add(Активация("relu"))
49         model.add (Пакетная нормализация (ось = chanDim))
50         model.add(Conv2D(384, (3, 3), padding="то же самое",
51                           kernel_regularizer=l2(reg)))
52         model.add(Активация("relu"))
53         model.add (Пакетная нормализация (ось = chanDim))
54         model.add(Conv2D(256, (3, 3), padding="то же самое",
55                           kernel_regularizer=l2(reg)))
56         model.add(Активация("relu"))
57         model.add (Пакетная нормализация (ось = chanDim))
58         model.add(MaxPooling2D(pool_size=(3, 3), шаги=(2, 2)))
59         model.add (Выпадение (0,25))

```

Первые два фильтра CONV изучают 384 фильтра 3×3 , а третий фильтр CONV запоминает 256 фильтров 3×3 .
Опять же, наложение нескольких слоев CONV => RELU друг на друга перед применением деструктивного
Слой POOL позволяет нашей сети изучать более богатые и потенциально более отличительные функции.
Оттуда мы сворачиваем наше многомерное представление в стандартную прямую связь.

сеть с использованием двух полно связных слоев (по 4096 узлов в каждом):

```

61     # Блок №4: первый набор слоев FC => RELU
62     model.add(Свести())
63     model.add(Dense(4096, kernel_regularizer=l2(reg)))
64     model.add(Активация("relu"))
65     model.add(Пакетная нормализация())
66     model.add (Выпадение (0,5))

67
68     # Блок №5: второй набор слоев FC => RELU
69     model.add(Dense(4096, kernel_regularizer=l2(reg)))
70     model.add(Активация("relu"))
71     model.add(Пакетная нормализация())
72     model.add (Выпадение (0,5))

```

Пакетная нормализация применяется после каждой активации в наборах слоев FC, как и в CONV.
слои выше. Dropout, с большей вероятностью 50 процентов, применяется после каждого набора слоев FC, т.к.
является стандартным для подавляющего большинства CNN.

Наконец, мы определяем классификатор softmax, используя желаемое количество классов, и возвращаем
результатирующую модель для вызывающей функции:

```

74     # классификатор softmax
75     model.add(Dense(классы, kernel_regularizer=l2(reg)))
76     model.add(Активация("softmax"))

77
78     # вернуть построенную сетевую архитектуру
79     модель возврата

```

Как видите, внедрение AlexNet — довольно простой процесс, особенно когда вы
иметь «чертеж» архитектуры, представленный в таблице 10.1 выше. Всякий раз при реализации
архитектуры из публикаций, попробуйте посмотреть, предоставляют ли они такую таблицу, поскольку она делает реализацию

намного легче. Для ваших собственных сетевых архитектур используйте главу 19 Starter Bundle, посвященную визуализации сетевых архитектур, чтобы убедиться, что размеры входных и выходных томов соответствуют вашим ожиданиям.

10.4 Обучение AlexNet на Kaggle: Dogs vs. Cats

Теперь, когда архитектура AlexNet определена, давайте применим ее к задаче Kaggle Dogs vs. Cats . Откройте новый файл, назовите его train_alexnet.py и вставьте следующий код:

```

1 # импортировать необходимые пакеты
2 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом режиме
3 import matplotlib 4 matplotlib.use("Agg")

5
6 # импортируем необходимые пакеты 7
из конфига importdogs_vs_cats_config as config 8 из
pyimagesearch.preprocessing import ImageToArrayPreprocessor 9 из
pyimagesearch.preprocessing import SimplePreprocessor 10 из
pyimagesearch.preprocessing import PatchPreprocessor 11 из pyimagesearch.preprocessing
import MeanPreprocessor 12 из pyimagesearch.callbacks1 import Training fromMonitor
pyimagesearch.io импортирует HDF5DatasetGenerator 14 из pyimagesearch.nn.conv
импортирует AlexNet 15 из keras.preprocessing.image импортирует ImageDataGenerator
16 из keras.optimizers импортирует Adam 17 импортирует json 18 импортирует os

```

Строки 3 и 4 импортируют matplotlib, обеспечивая при этом настройку серверной части таким образом, чтобы мы могли сохранять цифры и графики на диск по мере обучения нашей сети. Затем мы реализуем наши препроцессоры в строках 8-11. Затем HDF5DatasetGenerator импортируется в строку 13 , поэтому мы можем получить доступ к пакетам обучающих данных из нашего сериализованного набора данных HDF5. AlexNet также реализует линию 14.

Наш следующий блок кода обрабатывает инициализацию нашего генератора дополнений данных через класс ImageDataGenerator:

```

20 # построить генератор обучающих изображений для увеличения данных 21
aug = ImageDataGenerator(rotation_range=20, zoom_range=0.15,
22     width_shift_range = 0,2, height_shift_range = 0,2, shear_range = 0,15, horizontal_flip
23     = True, fill_mode = «ближайший»)

```

Теперь давайте потратим время на инициализацию каждого из наших препроцессоров изображений:

```

25 # загрузить средства RGB для тренировочного
набора 26 mean = json.loads(open(config.DATASET_MEAN).read())
27
28 # инициализируем препроцессоры
изображений 29 sp = SimplePreprocessor(227, 227)
30 pp = PatchPreprocessor(227, 227) 31 mp =
MeanPreprocessor(означает ["R"], означает ["G"], означает ["B"]) 32 iap =
ImageToArrayPreprocessor()

```

В строке 26 мы загружаем сериализованные средства RGB с диска — это средства для каждого из соответствующих красных, зеленых и синих каналов в нашем наборе обучающих данных. Эти значения позже будут переданы в MeanPreprocessor для нормализации среднего вычитания.

В строке 29 создается экземпляр SimplePreprocessor , используемый для уменьшения размера входного изображения до 227×227 пикселей. Этот препроцессор будет использоваться в генераторе данных проверки, так как наши входные изображения имеют размер 256×256 пикселей; однако AlexNet предназначен для обработки только изображений 227×227 (поэтому нам нужно изменить размер изображения во время проверки).

Строка 30 создает экземпляр PatchPreprocessor — этот препроцессор будет случайным образом выбирать области 227×227 из входных изображений 256×256 во время обучения, выступая в качестве второй формы увеличения данных.

Затем мы инициализируем MeanPreprocessor в строке 31 , используя наши соответствующие средние значения Red, Green и Blue. Наконец, препроцессор ImageToArrayPreprocessor (строка 32) используется для преобразования изображений в массивы, совместимые с Keras.

Учитывая наши препроцессоры, давайте определим HDF5DatasetGenerator как для данных обучения, так и для данных проверки:

```
34 # инициализировать генераторы наборов данных для обучения и
  проверки 35 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 128, aug=aug,
36     препроцессоры = [pp, mp, iap], классы = 2) 37
  valGen = HDF5DatasetGenerator (config.VAL_HDF5, 128,
38     препроцессоры = [sp, mp, iap], классы = 2)
```

Строки 35 и 36 создают наш генератор набора обучающих данных. Здесь мы указываем путь к нашему обучающему файлу HDF5, указывая, что мы должны использовать размеры пакетов из 128 изображений, увеличение данных и три препроцессора: исправление, среднее значение и изображение в массив соответственно.

Строки 37 и 38 отвечают за создание экземпляра тестового генератора. На этот раз мы укажем путь к файлу HDF5 проверки, используем размер пакета 128, без увеличения данных и простой препроцессор, а не препроцессор исправления (поскольку увеличение данных не применяется к данным проверки).

Наконец, мы готовы инициализировать оптимизатор Adam и архитектуру AlexNet:

```
40 # инициализируем оптимизатор
41 print("[INFO] компилируем модель...") 42
  opt = Adam(lr=1e-3) 43 model =
  AlexNet.build(width=227, height=227, depth=3, классы =2, reg=0.0002)
44     45 model.compile(loss="binary_crossentropy",
  оптимизатор=opt, metrics=["точность"])
46
47
48 # построить набор обратных вызовов
49 путь = os.path.sep.join([config.OUTPUT_PATH, "{}.png".format( os.getpid())])
50     51 обратный вызов = [TrainingMonitor(путь)]
```

В строке 42 мы создаем экземпляр оптимизатора Adam, используя скорость обучения по умолчанию 0,001. Причина , по которой я выбрал Адама для этого эксперимента (а не SGD), двояка: 1. Я хотел дать вам представление об использовании более продвинутых оптимизаторов, которые мы рассмотрели в главе .

7.

2. Адам справляется с этой задачей классификации лучше, чем SGD (о чем я знаю из множества предыдущие эксперименты, которые я проводил до публикации этой книги).

Затем мы инициализируем AlexNet в строках 43 и 44, указывая, что каждое входное изображение будет иметь ширину 227 пикселей, высоту 227 пикселей, 3 канала, а сам набор данных будет иметь два класса (один для собак, а другой для кошек) . Мы также применим небольшой штраф за регуляризацию в размере 0,0002 , чтобы помочь бороться с переоснащением и повысить способность нашей модели обобщать набор для тестирования.

Мы будем использовать бинарную кросс-энтропию, а не категориальную кросс-энтропию (строки 45 и 46) , поскольку это проблема классификации только двух классов. Мы также определим обратный вызов TrainingMonitor в строке 51 , чтобы мы могли отслеживать производительность нашей сети по мере ее обучения.

Говоря об обучении сети, давайте сделаем это сейчас:

```
53 # обучаем сеть
54
55
56 model.fit_generator(trainGen.generator(),
57     steps_per_epoch=trainGen.numImages // 128,
58     validation_data=valGen.generator(),
59     validation_steps=valGen.numImages // 128, epochs
60     =75, max_queue_size=128 * 2, callbacks=callbacks ,
61     подробно=1)
```

Чтобы обучить AlexNet на наборе данных Kaggle Dogs vs. Cats с помощью нашего HDF5DatasetGenerator, нам нужно использовать метод модели fit_generator . Во-первых, мы передаем trainGen.generator() , генератор HDF5, используемый для создания мини-пакетов обучающих данных (строка 55). Чтобы определить количество пакетов в эпоху, мы делим общее количество изображений в обучающем наборе на размер нашего пакета (строка 56). Мы делаем то же самое в строках 57 и 58 для данных проверки. Наконец, укажем, что AlexNet будет обучаться на 75 эпох.

Последний шаг — просто сериализовать нашу модель в файл после обучения, а также закрыть каждый из обучающих и тестовых наборов данных HDF5 соответственно:

```
63 # сохранить модель в файл 64 print("[INFO]
65 сериализующая модель...") 65 model.save(config.MODEL_PATH,
66 overwrite=True)
66
67 # закрыть наборы данных HDF5 68 trainGen.close()
69 valGen.close()
```

Чтобы обучить AlexNet на наборе данных Kaggle Dogs vs. Cats, выполните следующую команду:

```
$ python train_alexnet.py
Эпоха 73/75 415 с - потеря:
0,4862 - акк: 0,9126 - val_loss: 0,6826 - val_acc: 0,8602 Эпоха 74/75 408 с - потеря:
0,4865 - акк: 0,9166 - val_loss: 0,6895/val_acc2: 0 75 401 с - потеря: 0,4813 - акк:
0,9166 - val_loss: 0,4195 - val_acc: 0,9297 [INFO] сериализующая модель...
```

График потерь/точности при обучении и проверке за 75 эпох можно увидеть на рис. 10.4. В целом мы видим, что графики обучения и точности хорошо коррелируют друг с другом, хотя мы могли бы помочь стабилизировать вариации потерь при проверке к концу цикла 75 эпох, применив



Рисунок 10.4: Обучение AlexNet на соревновании Kaggle Dogs vs. Cats, где мы получили 92,97% точность классификации на нашем проверочном наборе. Наша кривая обучения стабильна с изменениями в обучении точность/потери отражаются в соответствующем разделении проверки.

небольшое снижение скорости обучения. Изучение классификационного отчета AlexNet о собаках и кошках. набор данных, мы видим, что наши полученные результаты составляют 92,97% на проверочном наборе.

В следующем разделе мы оценим AlexNet на тестовом наборе, используя стандартный метод и метод избыточной выборки. Как продемонстрируют наши результаты, использование избыточной выборки может увеличить классификация от 1-3% в зависимости от вашего набора данных и сетевой архитектуры.

10.5 Оценка AlexNet

Чтобы оценить AlexNet на тестовом наборе, используя как наш стандартный метод, так и технику избыточной выборки, давайте создадим новый файл с именем crop_accuracy.py:

```
--- собаки_vs_коты
| --- конфигурация
| | --- build_dogs_vs_cats.py
| | --- crop_accuracy.py
| | --- Extract_features.py
| | --- train_alexnet.py
| | --- train_model.py
| | --- вывод
```

Оттуда откройте crop_accuracy.py и вставьте следующий код:

1 # импортируем необходимые пакеты

2 из конфигурации импортировать dogs_vs_cats_config как конфигурацию

```
3 из pyimagesearch.preprocessing import ImageToArrayPreprocessor 4 из
pyimagesearch.preprocessing import SimplePreprocessor 5 из pyimagesearch.preprocessing
import MeanPreprocessor 6 из pyimagesearch.preprocessing import CropPreprocessor
7 из pyimagesearch.io import HDF5DatasetGenerator 8 из pyimagesearch.utils.rank
import rank5_accuracy.9s ____ load_model 10 импортировать numpy как np 11
импортировать индикатор выполнения 12 импортировать json
```

Строки 2-12 импортируют необходимые пакеты Python. Стока 2 импортирует наш файл конфигурации Python для испытания Dogs vs. Cats. Мы также импортируем наши препроцессоры изображений в строках 3-6, включая ImageToArrayPreprocessor, SimplePreprocessor, MeanPreprocessor и CropPreprocessor. HDF5DatasetGenerator требуется, чтобы мы могли получить доступ к тестовому набору нашего набора данных и получить прогнозы по этим данным, используя нашу предварительно обученную модель.

Теперь, когда наш импорт завершен, давайте загрузим средства RGB с диска, инициализируем наше изображение препроцессоры и загрузить предварительно обученную сеть AlexNet:

```
14 # загрузить средства RGB для тренировочного
набора 15 mean = json.loads(open(config.DATASET_MEAN).read())
16
17 # инициализируем препроцессоры
изображений 18 sp = SimplePreprocessor(227, 227)
19 mp = MeanPreprocessor(means["R"], означает["G"], означает["B"]) 20 cp =
CropPreprocessor(227, 227) 21 iap = ImageToArrayPreprocessor()

22
23 # загрузить предварительно обученную сеть 24 print("[INFO]
loading model...") 25 model = load_model(config.MODEL_PATH)
```

Прежде чем мы применим избыточную выборку и 10-кадрирование, давайте сначала получим базовый уровень на тестовом наборе. используя только исходное тестовое изображение в качестве входных данных для нашей сети:

```
27 # инициализируем генератор тестовых наборов данных, затем делаем прогнозы 28 #
тестовых данных 29 print("[INFO] прогнозируем тестовые данные (без посевов...)") 30
testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64, preprocessors= [сп, мп, иап],
классы=2)
31
32 прогноза = model.predict_generator(testGen.generator(),
33     steps=testGen.numImages // 64, max_queue_size=64 * 2)
34
35 # вычислить точность rank-1 и rank-5 36 (rank1, _) =
rank5_accuracy(predictions, testGen.db["labels"]) 37 print("[INFO] rank-1: {:.2f}
%" .format(rank1 * 100)) 38 testGen.close()
```

Строки 30 и 31 инициализируют HDF5DatasetGenerator для доступа к тестовому набору данных в пакетах по 64 изображения. Поскольку мы получаем базовый уровень, мы будем использовать только SimplePreprocessor для изменения размера входных изображений 256× 256 до 227× 227 пикселей с последующей нормализацией среднего и

преобразование пакета в массив изображений, совместимый с Keras. Строки 32 и 33 затем используют генератор для оценки AlexNet в наборе данных.

Учитывая наши прогнозы, мы можем вычислить нашу точность на тестовом наборе (строки 36-38). Уведомление здесь, как мы заботимся только о точности ранга 1, потому что собаки против кошек - это 2-й класс набор данных — вычисление точности 5-го ранга для набора данных с двумя классами тривиально сообщает о 100 процентах точность классификации.

Теперь, когда у нас есть основа для стандартной техники оценки, давайте перейдем к избыточной выборке:

```
40 # повторно инициализировать генератор набора тестов, на этот раз исключая
41 # 'Простой препроцессор'
42 testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64,
43     препроцессоры=[mp], классы=2)
44 прогноза = []
45
46 # инициализировать индикатор выполнения
47 widgets = ["Оценка:", progressbar.Percentage(), " ",
48             progressbar.Bar(), " ", progressbar.ETA()]
49 pbar = progressbar.ProgressBar(maxval=testGen.numImages // 64,
50     виджеты=виджеты).start()
```

В строках 42 и 43 мы повторно инициализируем HDF5DatasetGenerator, на этот раз указывая ему используйте только MeanPreprocessor — позже мы применим как передискретизацию, так и преобразование массива Keras в трубопроводе. Строки 47-50 также инициализируют виджеты индикатора прогресса на нашем экране, если нам это интересно. в отображении прогресса оценки на нашем экране.

Учитывая повторный экземпляр testGen, теперь мы готовы применить технику 10-обрезки:

```
52 # цикл по одному проходу тестовых данных
53 для (i, (изображения, метки)) в enumerate(testGen.generator(passes=1)):
54     # цикл по каждому отдельному изображению
55     для изображения в изображениях:
56         # применяем к изображению препроцессор обрезки, чтобы сгенерировать 10
57         # разделяем обрезки, затем конвертируем их из изображений в массивы
58         урожай = cp.preprocessing(изображение)
59         урожай = np.array([iap.preprocessing(c) для c в урожаях],
60                           dtype="поплавок32")
61
62         # делаем прогнозы по урожаям, а затем усредняем их
63         # вместе, чтобы получить окончательный прогноз
64         pred = model.predict(урожай)
65         предсказания .append (пред. среднее (ось = 0))
66
67     # обновить индикатор выполнения
68     pbar.update(я)
```

В строке 53 мы начинаем перебирать каждую партию изображений в тестовом генераторе. Обычно HDF5DatasetGenerator настроен на бесконечный цикл, пока мы явно не скажем ему остановиться (обычно установка максимального количества итераций через Keras при обучении); однако, поскольку мы сейчас оценивая, мы можем поставить pass=1, чтобы указать, что тестовые данные нужно зациклить только один раз.

Затем для каждого изображения в пакете изображений (строка 55) мы применяем препроцессор 10-кадрирования на Страна 58, которая преобразует изображение в массив из десяти изображений 227× 227 . Эти кадры 227× 227 были извлечены из исходной партии 256× 256 на основе:

- Верхний левый угол
- Верхний правый угол
- Нижний правый угол •
- Нижний левый угол •

Соответствующие горизонтальные

перевороты После того, как у нас есть обрезки, мы пропускаем их через модель в строке 64 для предсказания.

Окончательный прогноз (строка 65) представляет собой среднее значение вероятностей для всех десяти культур.

Наш последний блок кода отображает точность метода передискретизации:

```
70 # вычислить точность ранга 1 71
pbar.finish() 72 print("[INFO] предсказание
на тестовых данных (с посевами)...") 73 (rank1, _) = rank5_accuracy(predictions,
testGen.db[" метки"]) 74 print("[INFO] rank-1: {:.2f}%".format(rank1 * 100)) 75 testGen.close()
```

Чтобы оценить AlexNet в наборе данных Kaggle Dog vs. Cats, просто выполните следующую команду:

```
$ python crop_accuracy.py [INFO]
модель загрузки...
[INFO] прогнозирование по тестовым данным (без посевов)...
[ИНФО] Ранг-1: 92,60%
Оценка: 100% ##### | Время: 0:01:12 [INFO]
предсказание на тестовых данных (с посевами)...
[ИНФО] ранг-1: 94,00%
```

Как показывают наши результаты, мы достигаем точности 92,60% на тестовом наборе. Однако, применяя метод избыточной выборки с 10 кадрами, мы можем повысить точность классификации до 94,00%, увеличившись на 1,4%, что было достигнуто просто путем взятия нескольких кадров входного изображения и усреднения результатов. Этот простой и несложный трюк — простой способ заработать несколько дополнительных процентов при оценке вашей сети.

10.6 Получение места в топ-5 в таблице лидеров Kaggle

Конечно, если

бы вы посмотрели на таблицу лидеров Kaggle Dogs vs. Cats, вы бы заметили, что даже для того, чтобы пробиться в топ-25, нам потребуется точность 96,69%, что является нашим текущий метод не способен достичь. Итак, каково решение?

Ответ заключается в передаче обучения, в частности, в передаче обучения посредством извлечения признаков. Хотя набор данных ImageNet состоит из 1000 категорий объектов, значительная часть из них включает как виды собак, так и виды кошек. Таким образом, сеть, обученная на ImageNet, может не только сказать вам, было ли на изображении изображение собаки или кошки, но и узнать, какой именно породы это животное. Учитывая, что сеть, обученная на ImageNet, должна быть в состоянии различать таких мелких животных, естественно предположить, что функции, извлеченные из предварительно обученной сети, вероятно, хорошо подходят для того, чтобы претендовать на первое место в Kaggle Dogs vs. Cats. таблица лидеров.

Чтобы проверить эту гипотезу, давайте сначала извлечем функции из предварительно обученной архитектуры ResNet и затем обучите классификатор логистической регрессии поверх этих функций.

10.6.1 Извлечение функций с помощью ResNet

Метод переноса обучения с помощью метода извлечения признаков, который мы будем использовать в этом разделе, в значительной степени основан на главе 3. Я рассмотрю все содержимое extract_features.py для полноты картины;

однако, пожалуйста, обратитесь к Главе 3, если вам нужны дополнительные знания по извлечению признаков с использованием CNN.

Для начала откройте новый файл, назовите его extract_features.py и вставьте следующий код:

```
1 # импортировать необходимые пакеты
2 из keras.applications import ResNet50 3 из
keras.applications import imagenet_utils 4 из
keras.preprocessing.image import img_to_array 5 из
keras.preprocessing.image import load_img 6 из sklearn.preprocessing
import LabelEncoder 7 из pyimagesearch.io импортировать
HDF5DatasetWriter 8 из путей импорта imutils 9 импортировать
пимпу как np 10 импортировать индикатор выполнения 11
импортировать argparse 12 импортировать случайные 13
импортировать ОС
```

Строки 2-13 импортируют необходимые пакеты Python. Мы импортируем класс ResNet50 в строку 2 , чтобы получить доступ к предварительно обученной архитектуре ResNet. Мы также будем использовать HDF5DatasetWriter в строке 7 , чтобы мы могли записывать извлеченные функции в эффективный формат файла HDF5.

Оттуда 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 входному набору
данных") 19 ap.add_argument("-o", "--output", required=True,
20     help="путь к выходному файлу HDF5")
21 ap.add_argument("-b", "--batch-size", type=int, default=16,
22     help="размер пакета изображений для передачи по сети") 23
ap.add_argument("-s", "--buffer-size", type=int, default=1000, help="размер буфера
извлечения объектов") 25 аргументов = переменные (ap.parse_args())
26
27 # сохранить размер партии в удобной переменной
28 бит = аргументы["размер_пакета"]
```

Здесь нам нужны только два обязательных аргумента командной строки: --dataset, который представляет собой путь к входному набору данных изображений Dogs vs. Cats, а также --output, путь к выходному файлу HDF5, содержащему функции, извлеченные через ResNet.

Затем давайте возьмем пути к изображениям Dogs vs. Cats, находящимся на диске, и затем используем файл пути для извлечения имен меток:

```
30 # получить список изображений, которые мы будем описывать, а затем случайным
образом 31 # перетасовать их, чтобы облегчить обучение и тестирование разделения
с помощью 32 # нарезки массива во время обучения 33 print("[INFO] loading images...")
34 imagePaths = list(paths.list_images(аргументы["набор данных"])) 35
random.shuffle(imagePaths)
```

```
36
37 # извлечь метки классов из путей к изображениям, затем закодировать
38 # ярлыки
39 меток = [p.split(os.path.sep)[-1].split(".")[0] для p в imagePaths]
40 le = LabelEncoder()
41 метка = le.fit_transform(метки)
```

Теперь мы можем загрузить наши предварительно обученные веса ResNet50 с диска (исключая слои FC):

```
43 # загрузить сеть ResNet50
44 print("[ИНФО] загрузка сети...")
45 модель = ResNet50 (веса = "imagenet", include_top = False)
```

Чтобы сохранить функции, извлеченные из ResNet50, на диск, нам нужно создать экземпляр
объект HDF5DatasetWriter:

```
47 # инициализировать средство записи набора данных HDF5, затем сохранить метку класса
48 # имен в наборе данных
49 набор данных = HDF5DatasetWriter((len(imagePaths), 2048),
50         args["output"], dataKey="features", bufSize=args["buffer_size"])
51 набор данных.storeClassLabels(le.classes_)
```

Окончательный средний слой пула ResNet50 — 2048-d, поэтому мы указываем значение 2048 .
в качестве размерности для нашего HDF5datasetWriter.

Мы также инициализируем индикатор выполнения, чтобы отслеживать процесс извлечения признаков:

```
53 # инициализировать индикатор выполнения
54 widgets = ["Извлечение признаков: ", progressbar.Percentage(), " ",
55             progressbar.Bar(), " ", progressbar.ETA()]
56 pbar = progressbar.ProgressBar (maxval = len (imagePaths),
57                                 виджеты=виджеты).start()
```

Извлечение признаков из набора данных с помощью CNN такое же, как в главе 3. Во-первых, мы
цикл по imagePaths в пакетах:

```
59 # цикл по изображениям в пакетах
60 для i в np.arange(0, len(imagePaths), bs):
61     # извлечь пакет изображений и меток, затем инициализировать
62     # список актуальных изображений, которые будут передаваться по сети
63     # для извлечения признаков
64     batchPaths = imagePaths[i:i + bs]
65     batchLabels = метки[i:i + bs]
66     пакетные изображения = []
```

Затем следует предварительная обработка каждого изображения:

```
68     # цикл по изображениям и меткам в текущем пакете
69     для (j, imagePath) в перечислении (batchPaths):
70         # загружаем входное изображение с помощью вспомогательной утилиты Keras
```

```

71      # при изменении размера изображения до 224x224 пикселей
72      изображение = load_img (imagePath, target_size = (224, 224))
73      изображение = img_to_array (изображение)
74
75      # предварительно обработать изображение путем (1) увеличения размеров и
76      # (2) вычитание средней интенсивности пикселя RGB из
77      # Набор данных ImageNet
78      изображение = np.expand_dims (изображение, ось = 0)
79      изображение = imagenet_utils.preprocess_input(изображение)
80
81      # добавляем изображение в пакет
82      batchImages.append(изображение)

```

А затем пропуская пакетные изображения через сетевую архитектуру, что позволяет нам извлекать функции из последнего уровня POOL ResNet50:

```

84      # передавать изображения по сети и использовать выходные данные как
85      # наши актуальные возможности
86      пакетные изображения = np.vstack (пакетные изображения)
87      функции = model.predict (batchImages, batch_size = bs)
88
89      # изменить форму объектов так, чтобы каждое изображение было представлено
90      # сглаженный вектор признаков выходных данных 'MaxPooling2D'
91      функции = функции.изменить((особенности.форма[0], 2048))

```

Эти извлеченные функции затем добавляются в наш набор данных:

```

93      # добавляем функции и метки в наш набор данных HDF5
94      dataset.add (функции, пакетные метки)
95      pbar.update(я)
96
97 # закрыть набор данных
98 набор данных.close()
99 пбар.finish()

```

Чтобы использовать ResNet для извлечения функций из набора данных Dogs vs. Cats, просто выполните команду следующая команда:

```
$ python extract_features.py --dataset ..../datasets/kaggle_dogs_vs_cats/train \
--output ..../наборы данных/kaggle_dogs_vs_cats/hdf5/features.hdf5
[INFO] загрузка изображений...
[INFO] загрузка сети...
Особенности извлечения: 100% ##### | Время: 0:06:18
```

После завершения выполнения команды у вас должен появиться файл с именем `dogs_vs_cats_features.hdf5`. в вашем выходном каталоге:

```
$ ls -l вывод/dogs_vs_cats_features.hdf5
-rw-rw-r-- Адриан 409806272 3 июня 07:17 output/dogs_vs_cats_features.hdf5
```

Учитывая эти функции, мы можем обучить классификатор логистической регрессии поверх них, чтобы (в идеале) получить пятерку лучших в таблице лидеров Kaggle Dogs vs. Cats.

10.6.2 Обучение классификатора логистической регрессии

Чтобы обучить наш классификатор логистической регрессии, откройте новый файл и назовите его `train_model.py`. Оттуда мы можем начать:

```
1 # импортируем необходимые пакеты 2
из sklearn.linear_model импортируем LogisticRegression 3 из
sklearn.model_selection import GridSearchCV 4 из sklearn.metrics
import classification_report 5 из sklearn.metrics import precision_score
6 import argparse 7 import pickle 8 import h5py
```

Строки 2-8 импортируют необходимые пакеты Python. Затем мы проанализируем наши аргументы командной строки:

```
10 # построить разбор аргумента и разобрать аргументы 11 ap =
argparse.ArgumentParser() 12 ap.add_argument("-d", "--db", required=True,
help="path HDF5 database") 13
14 ap.add_argument("-m", "--model", required=True,
help="путь к выходной модели") 15
ap.add_argument("-j", "--jobs", type=int, default=-1,
16           help="# заданий для запуска при настройке гиперпараметров")
17
18 args = vars(ap.parse_args())
```

Здесь нам нужны только два переключателя, путь к входу HDF5 `--db` вместе с путем к вывод логистической регрессии `--model` после завершения обучения.

Далее откроем набор данных HDF5 для чтения и определим разделение обучения и тестирования — 75% данных для обучения и 25% для тестирования:

```
20 # открыть базу данных HDF5 для чтения, затем определить индекс 21 # разделения
обучения и тестирования при условии, что эти данные были 22 # уже перемешаны
*перед* записью на диск 23 db = h5py.File(args["db"], "r") 24 i = int(db["labels"].shape[0]
* 0,75)
```

Учитывая это разделение функций, мы выполним поиск в сетке по гиперпараметру С классификатора `LogisticRegression`:

```
26 # определяем набор параметров, которые мы хотим настроить, затем начинаем
27 # поиск по сетке, где мы оцениваем нашу модель для каждого значения С 28
print("[INFO] настройки гиперпараметров...") 29 params = {"C" : [0,001, 0,001, 0,01, 0,1,
1,0]} 30 модель = GridSearchCV(LogisticRegression(), params, cv=3,
31           n_jobs=args["jobs"]) 32
модель.fit(db["features"][:i], db["labels"][:i]) 33 print("[INFO] лучшие
гиперпараметры: {}".format(модель.best_params_))
```

Как только мы нашли лучший выбор С, мы можем создать отчет о классификации для тестирования.

набор:

```

35 # создать отчет о классификации для модели
36 print("[INFO] оценка...")
37 pres = model.predict(db["features"][:i:])
38 print(classification_report(db["labels"][:i:], pres,
39     target_names=db["label_names"]))
40
41 # вычислить необработанную точность с дополнительной точностью
42 acc = показатель_точности(дб["метки"][:i:], преды)
43 print("[INFO] оценка: {}".format(acc))

```

И, наконец, обученную модель можно сериализовать на диск для дальнейшего использования, если мы того пожелаем:

```

45 # сериализовать модель на диск
46 print("[INFO] сохранение модели...")
47 f = открыть(args["модель"], "wb")
48 f.write(pickle.dumps(model.best_estimator_))
49 ф.закрыть()
50
51 # закрыть базу
52 дБ.close()

```

Чтобы обучить нашу модель функциям ResNet50, просто выполните следующую команду:

```

python train_model.py --db ../datasets/kaggle_dogs_vs_cats/hdf5/features.hdf5 \
    --model dogs_vs_cats.pickle
[INFO] настройка гиперпараметров...
[INFO] лучшие гиперпараметры: {'C': 0,001}
[INFO] оценка...
      точность      вспомнить поддержку f1-score
Кот        0,99        0,98        0,99        3160
собака      0,98        0,99        0,99        3090
среднее / общее    0,99        0,99        0,99        6250
[ИНФО] оценка: 0,98688
[INFO] сохранение модели...

```

Как видно из вывода, наш подход к использованию трансферного обучения через извлечение признаков дает впечатляющую точность 98,69%, достаточную для того, чтобы занять второе место в рейтинге Kaggle Dogs. против кошек лидеров.

10.7 Резюме

В этой главе мы глубоко погрузились в набор данных Kaggle Dogs vs. Cats и изучили методы получить > 90% точность классификации на нем:

1. Обучение AlexNet с нуля.
2. Применение трансферного обучения через ResNet.

Архитектура AlexNet — основополагающая работа, впервые представленная Крижевским и др. в 2012 году [6]. Используя нашу реализацию AlexNet, мы достигли 94-процентной точности классификации. Это очень респектабельная точность, особенно для сети, обученной с нуля. Дальнейшая точность, вероятно, может быть получено:

1. Получение большего количества обучающих данных.
2. Применение более агрессивной аугментации данных.
3. Углубление сети.

Однако полученных нами 94 процентов недостаточно даже для того, чтобы пробиться в топ-25 лидеров, не говоря уже о топ-5. Таким образом, чтобы получить наше место в пятерке лучших, мы полагались на трансферное обучение посредством извлечения признаков, в частности, на архитектуру ResNet50, обученную на наборе данных ImageNet. Поскольку ImageNet содержит множество примеров пород как собак, так и кошек, применение предварительно обученной сети к этой задаче является естественным и простым методом, позволяющим получить более высокую точность с меньшими усилиями. Как показали наши результаты, мы смогли получить точность классификации 98,69% , что достаточно для того, чтобы претендовать на вторую позицию в таблице лидеров Kaggle Dogs vs. Cats.

11. ГуглеНет

В этой главе мы изучим архитектуру GoogLeNet, представленную Szegedy et al. в своей статье 2014 года *Going Deeper With Convolutions* [17]. Эта статья важна по двум причинам.

Во-первых, архитектура модели крошечная по сравнению с AlexNet и VGGNet (28 МБ для самих весов). Авторы смогли добиться такого резкого уменьшения размера сетевой архитектуры (при этом увеличивая глубину всей сети), удалив полно связные слои и вместо этого используя глобальный усредненный пул. Большинство весов в CNN можно найти в плотных слоях FC — если эти слои можно удалить, экономия памяти будет огромной.

Во-вторых, Szegedy et al. бумага использует сеть в сети или микроархитектуре, когда построение общей макроархитектуры. До сих пор мы видели только последовательные нейронные сети, в которых выходные данные одной сети напрямую передаются следующей. Теперь мы увидим микроархитектуры, небольшие строительные блоки, которые используются внутри остальной архитектуры, где выходные данные одного уровня могут разделяться на несколько различных путей и объединяться позже.

В частности, Szegedy et al. предоставил сообществу глубокого обучения модуль Inception, строительный блок, который вписывается в сверточную нейронную сеть, позволяя ей изучать слои CONV с несколькими размерами фильтров, превращая модуль в многоуровневый экстрактор функций.

Микроархитектуры, такие как Inception, вдохновили другие важные варианты, включая модуль Residual в ResNet [24] и модуль Fire в SqueezeNet [32]. Мы обсудим модуль Inception (и его варианты) позже в этой главе. После того, как мы изучим модуль Inception и убедимся, что знаем, как он работает, мы затем реализуем уменьшенную версию GoogLeNet под названием «MiniGoogLeNet» — мы обучим эту архитектуру на наборе данных CIFAR-10 и получим более высокую точность, чем в любом другом наших предыдущих глав.

Оттуда мы перейдем к более сложному cs231n Tiny ImageNet Challenge [4]. Эта задача предлагается студентам, зачисленным в Стэнфордский класс cs231n Convolutional Neural Networks for Visual Recognition [39], как часть их финального проекта. Это означает дать им представление о проблемах, связанных с крупномасштабным глубоким изучением современных архитектур, не требуя столько времени и усилий, как работа со всем набором данных ImageNet.

Обучая GoogLeNet с нуля на Tiny ImageNet, мы продемонстрируем, как занять высшую позицию в таблице лидеров Tiny ImageNet. И в нашей следующей главе мы будем использовать ResNet.

претендовать на первое место среди моделей, обученных с нуля.

Давайте продолжим и начнем эту главу с обсуждения модуля Inception.

11.1 Начальный модуль (и его варианты)

Современные современные сверточные нейронные сети используют микроархитектуры, также называемые модулями «сеть в сети», первоначально предложенные Lin et al. [40]. Я лично предпочитаю термин микроархитектура, поскольку он лучше описывает эти модули как строительные блоки в контексте общей макроархитектуры (то есть того, что вы на самом деле создаете и обучаете).

Микроархитектуры — это небольшие строительные блоки, разработанные специалистами по глубокому обучению, позволяющие сетям обучаться (1) быстрее и (2) эффективнее, при этом увеличивая глубину сети. Эти строительные блоки микроархитектуры складываются вместе с обычными типами слоев, такими как CONV, POOL и т. д., для формирования общей макроархитектуры.

В 2014 году Szegedy et al. представил начальный модуль. Общая идея начала модуль двойной:

1. Может быть сложно определить размер фильтра, который вам нужно изучить на заданных слоях CONV . Должны ли они быть фильтрами 5×5 ? Как насчет фильтров 3×3 ? Должны ли мы изучать локальные особенности с помощью фильтров 1×1 ? Вместо этого, почему бы не изучить их все и позволить модели решить? Внутри начального модуля мы изучаем все три фильтра 5×5 , 3×3 и 1×1 (вычисляя их параллельно), объединяя полученные карты объектов по измерению канала. Следующий уровень в архитектуре GoogLeNet (который может быть еще одним начальным модулем) получает эти конкатенированные смешанные фильтры и выполняет тот же процесс. В целом этот процесс позволяет GoogLeNet изучать как локальные функции с помощью более мелких сверток, так и абстрактные функции с более крупными свертками — нам не нужно жертвовать нашим уровнем абстракции за счет более мелких функций.
2. Изучив несколько размеров фильтров, мы можем превратить модуль в многоуровневый экстрактор признаков. Фильтры 5×5 имеют больший восприимчивый размер и могут изучать больше абстрактных функций. Фильтры 1×1 по определению являются локальными. Фильтры 3×3 служат балансом между ними.

11.1.1 Начало

Теперь, когда мы обсудили мотивацию модуля Inception, давайте посмотрим на сам модуль на рис. 11.1.



Функция активации (ReLU) неявно применяется после каждого слоя CONV . Для экономии места эта функция активации не была включена в схему сети выше. Когда мы реализуем GoogLeNet, вы увидите, как эта активация используется в модуле Inception.

В частности, обратите внимание на то, как начальный модуль разветвляется на четыре отдельных пути от входного слоя. Первая ветвь в начальном модуле просто изучает набор локальных признаков 1×1 из входных данных.

Во второй партии сначала применяется свертка 1×1 не только как форма изучения локальных признаков, но и как уменьшение размерности. Большие свертки (т. е. 3×3 и 5×5) по определению требуют больше вычислений для выполнения. Следовательно, если мы сможем уменьшить размерность входных данных для этих более крупных фильтров, применяя свертки 1×1 , мы сможем уменьшить объем вычислений , требуемых нашей сетью. Следовательно, количество фильтров, изученных в преобразовании 1×1 во второй ветви, всегда будет меньше, чем количество фильтров 3×3 , изученных непосредственно после этого.

Третья ветвь использует ту же логику, что и вторая ветвь, только на этот раз с целью изучения фильтров 5×5 . Мы еще раз уменьшаем размерность с помощью сверток 1×1 , а затем подаем результат в фильтры 5×5 .

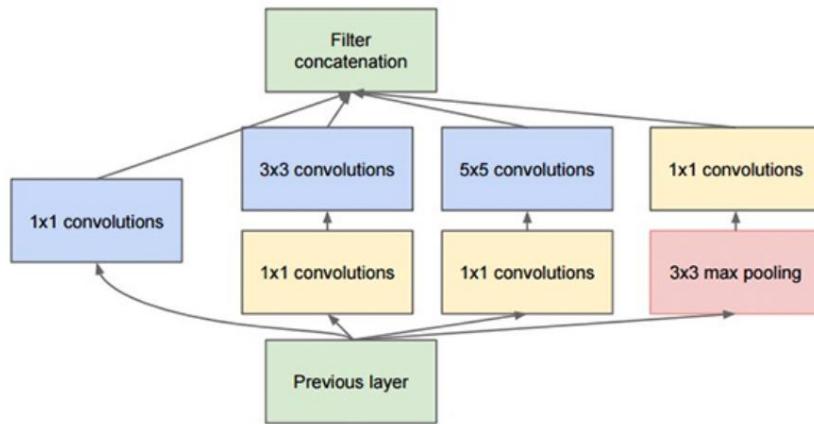


Рисунок 11.1: Исходный модуль Inception, используемый в GoogLeNet. Модуль Inception действует как «многоуровневый экстрактор признаков», вычисляя свертки 1×1 , 3×3 и 5×5 в одном и том же модуле сети. Рисунок из Szegedy et al., 2014 [17].

Четвертая и последняя ветвь начального модуля выполняет максимальное объединение 3×3 с шагом 1×1 — эту ветвь обычно называют ветвью проекции пула. Исторически сложилось так, что модели, выполняющие объединение, демонстрировали способность достигать более высокой точности, хотя теперь мы знаем благодаря работе Springenberg et al. в своей статье 2014 года «Стремление к простоте: вся сверточная сеть» [41] говорится, что это не обязательно верно, и что слои POOL можно заменить слоями CONV для уменьшения размера тома.

В случае Szegedy et al., этот уровень POOL был добавлен просто из-за того, что считалось, что они необходимы для разумной работы CNN. Затем выходные данные POOL передаются в другую серию сверток 1×1 для изучения локальных особенностей.

Наконец, все четыре ветви начального модуля сходятся в том месте, где они объединяются вместе по измерению канала. Особое внимание уделяется во время реализации (через заполнение нулями), чтобы гарантировать, что выходные данные каждой ветви имеют одинаковый размер тома, что позволяет объединять выходные данные. Затем выходные данные начального модуля передаются на следующий уровень в сети. На практике мы часто размещаем несколько начальных модулей друг над другом, прежде чем выполнять операцию объединения, чтобы уменьшить размер тома.

11.1.2 Миницепция

Конечно, исходный модуль Inception был разработан для GoogLeNet таким образом, чтобы его можно было обучить на наборе данных ImageNet (где предполагается, что каждое входное изображение имеет размер $224 \times 224 \times 3$) и получить современную точность. Для небольших наборов данных (с меньшими пространственными размерами изображения), где требуется меньше сетевых параметров, мы можем упростить начальный модуль.

Я впервые узнал о модуле «Minicception» из твита @ericjang11 (<https://twitter.com/ericjang11>) и @pluskid (<https://twitter.com/pluskid>) где они прекрасно визуализируют уменьшенный вариант Inception, используемый при обучении набора данных CIFAR-10 (рис. 11.2; спасибо @ericjang11 и @pluskid).

После небольшого исследования выяснилось, что этот рисунок был взят из публикации Чжана и др. 2017 года «Понимание глубокого обучения требует переосмыслиния обобщения» [42]. В верхней строке рисунка описаны три модуля, используемые в их реализации MiniGoogLeNet:

- Слева: модуль свертки, отвечающий за выполнение свертки, нормализацию пакетов, и активация.

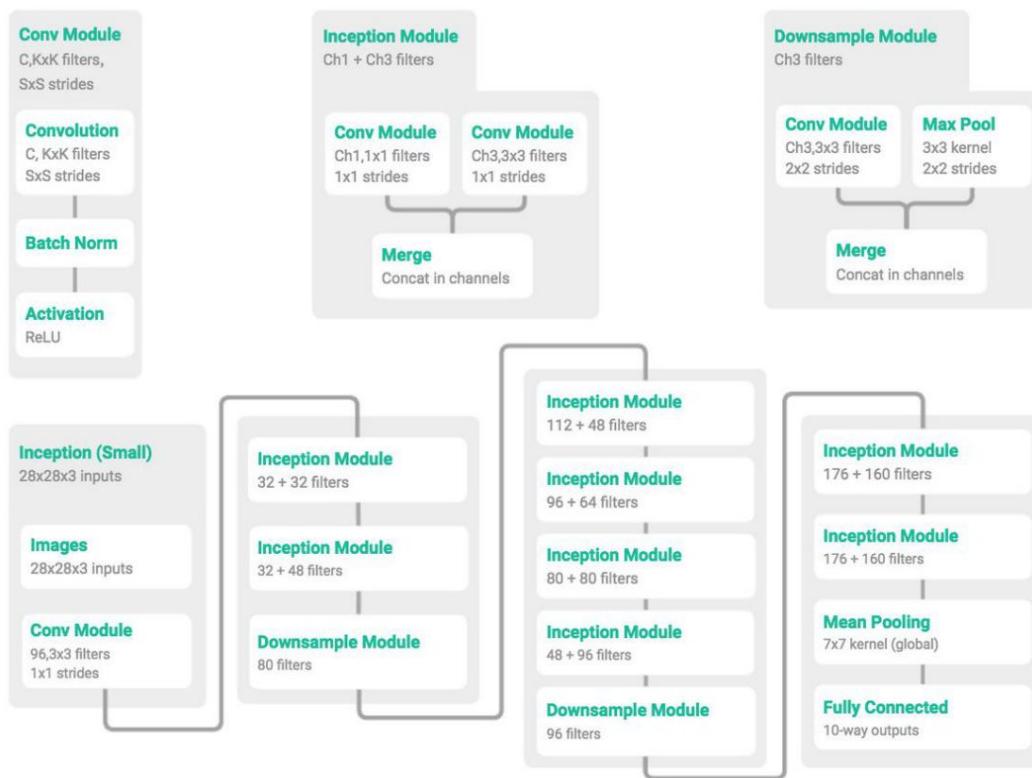


Рисунок 11.2: Архитектура Miniception состоит из строительных блоков, включая модуль свертки , модуль Inception и модуль Downsample. Эти модули объединяются в общую архитектуру.

- Средний: модуль Miniception, который выполняет два набора сверток, один для фильтров 1×1 , а другой для фильтров 3×3 , а затем объединяет результаты. Перед фильтром 3×3 уменьшение размерности не выполняется, поскольку (1) входные объемы уже будут меньше (поскольку мы будем использовать набор данных CIFAR-10) и (2) для уменьшения количества параметров в сети.
- Справа: модуль понижающей выборки, который применяет как свертку, так и максимальное объединение для уменьшения размерности, а затем выполняет конкатенацию по измерению фильтра.

Затем эти стандартные блоки используются для построения архитектуры MiniGoogLeNet в нижней строке. Здесь вы заметите, что авторы поместили пакетную нормализацию перед активацией (предположительно , потому, что это сделали Сегеди и др.), в отличие от того, что сейчас рекомендуется при реализации CNN.

В этой книге я остановился на реализации оригинальной работы автора, помещая пакетную нормализацию перед активацией, чтобы воспроизвести результаты. В ваших собственных экспериментах рассмотрите возможность замены этого порядка.

В следующем разделе мы реализуем архитектуру MiniGoogLeNet и применим ее к набору данных CIFAR-10. После этого мы будем готовы внедрить полный начальный модуль и решить задачу cs231n Tiny ImageNet.

11.2 MiniGoogLeNet на CIFAR-10

В этом разделе мы собираемся реализовать архитектуру MiniGoogLeNet с помощью модуля Miniception. Затем мы обучим MiniGoogLeNet на наборе данных CIFAR-10. Как покажут наши результаты,

эта архитектура обеспечит > 90% точности на CIFAR-10, что намного лучше, чем все наши предыдущие попытки.

11.2.1 Реализация MiniGoogLeNet

Для начала давайте сначала создадим файл с именем minigooglenet.py внутри модуля conv_pyimagesearch.nn — здесь будет жить наша реализация класса MiniGoogLeNet:

```
-- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ио
| |--- нн
| | |--- __init__.py
| | |--- конв.
| | |--- __init__.py
| | |--- alexnet.py
| | |--- lenet.py
| | |--- minigooglenet.py
| | |--- minivggnet.py
| | |--- fheadnet.py
| | |--- мелкая сеть.py
| |--- предварительная обработка
| |--- утилиты
```

Оттуда откройте minigooglenet.py и вставьте следующий код:

```
1 # импортируем необходимые пакеты
2 из keras.layers.normalization import BatchNormalization
3 из keras.layers.convolutional импорт Conv2D
4 из keras.layers.convolutional import AveragePooling2D
5 из keras.layers.convolutional импортировать MaxPooling2D
6 из keras.layers.core импорт Активация
7 из keras.layers.core import Dropout
8 из keras.layers.core импорт Плотный
9 из keras.layers import Flatten
10 из keras.layers import Input
11 из импорта keras.models Модель
12 из keras.layers import concatenate
13 из keras импортировать бэкэнд как K
```

Строки 2-13 импортируют необходимые пакеты Python. Вместо того, чтобы импортировать класс Sequential где выходные данные одного слоя направую передаются в следующий, вместо этого нам нужно будет использовать класс Model (строка 11). Использование Model вместо Sequential позволит нам создать сетевой график с разбиениями. и разветвления, как в модуле Inception. Еще один импорт, который вы еще не видели, — это конкатенация функция в строке 12. Как следует из названия, эта функция принимает набор входных данных и объединяет их вдоль заданной оси, которая в данном случае будет размерностью канала.

Мы будем реализовывать точную версию MiniGoogLeNet, как показано на рис. 11.2 выше. Итак, начнем с conv_module:

15 класс МиниГуглеНет:
16 @статический метод

```

17     def conv_module(x, K, kX, kY, шаг, chanDim, padding="same"):
18         # определить шаблон CONV => BN => RELU x = Conv2D(K,
19         # (kX, kY), strides=stride, padding=padding)(x) x = BatchNormalization(axis=chanDim)(x) x =
20         Activation(" относительно")(x)
21
22
23     # вернуть блок
24     вернуть x

```

Функция `conv_module` отвечает за применение свертки, за которой следует пакет нормализация, а затем, наконец, активация. Параметры метода подробно описаны ниже:

- `x`: входной слой для функции.
- `K`: количество фильтров, которые будут изучать наш слой CONV.
- `kX` и `kY`: Размер каждого из `K` фильтров, которые будут изучены.
- `шаг`: шаг слоя CONV.
- `chanDim`: размерность канала, полученная либо из «последние каналы», либо из «последние каналы».

первый заказ.

`padding`: тип заполнения, применяемого к слою CONV.

В строке 19 мы создаем сверточный слой. Фактические параметры `Conv2D` идентичны примерам в предыдущих архитектурах, таких как AlexNet и VGGNet, но здесь меняется способ подачи входных данных на данный уровень.

Поскольку для определения сетевой архитектуры мы используем `Model`, а не `Sequential`, мы не можем вызывать `model.add`, так как это означало бы, что выходные данные одного уровня последовательно переходят на следующий уровень. Вместо этого мы указываем входной слой в скобках в конце вызова функции, который называется функциональным API. Каждый экземпляр слоя в модели вызывается для тензора, а также возвращает тензор. Следовательно, мы можем предоставить входные данные для данного слоя, вызвав его как функцию после создания экземпляра объекта.

Шаблон для построения слоев таким образом можно увидеть ниже:

вывод = Слой (параметры) (ввод)

Найдите секунду, чтобы ознакомиться с этим новым стилем добавления слоев в сеть, поскольку мы использовать его всякий раз, когда мы определяем непоследовательные сети.

Выходные данные слоя `Conv2D` затем передаются в слой `BatchNormalization` в строке 20. Затем выходные данные `BatchNormalization` проходят через активацию `ReLU` (строка 21). Если бы мы построили фигуру, помогающую нам визуализировать `conv_module`, она выглядела бы так, как показано на рис. 11.3.

conv_module



Рисунок 11.3: Модуль `conv_module` архитектуры MiniGoogLeNet. Этот модуль не имеет ветвлений и представляет собой простой `CONV => BN => ACT`.

Сначала применяется свертка, затем нормализация партии, а затем активация. Обратите внимание, что этот модуль не выполнял никакого ветвления. Это изменится с определением `inception_module` ниже:

```

26     @статический метод
27     def inception_module(x, numK1x1, numK3x3, chanDim):
28         # определяем два модуля CONV, затем объединяем их
29         # размер канала
30         conv_1x1 = MiniGoogLeNet.conv_module(x, числоK1x1, 1,
31                                         (1, 1), чанДим)
32         conv_3x3 = MiniGoogLeNet.conv_module(x, числоK3x3, 3, 3,
33                                         (1, 1), чанДим)
34         x = объединить ([conv_1x1, conv_3x3], ось = chanDim)
35
36         # вернуть блок
37         вернуть x

```

Наш модуль Mininception будет выполнять два набора сверток — преобразование 1×1 и преобразование 3×3 . Эти две свертки будут выполняться параллельно, и результирующие функции будут объединены между собой. размерность канала.

В строках 30 и 31 используется удобный `conv_module`, который мы только что определили для изучения фильтров `numK1x1` (1×1). Затем в строках 32 и 33 снова применяется `conv_module` для изучения фильтров `numK3x3` (3×3). С помощью `conv_module`, мы можем повторно использовать код, и нам не нужно раздувать наш класс `MiniGoogLeNet`. путем вставки большого количества блоков `CONV => BN => RELU` блоков — об этом укладке заботятся лаконично через `conv_module`.

Обратите внимание, что входными данными для класса `Conv2D` 1×1 и 3×3 являются `x`, входные данные для слоя. Когда при использовании класса `Sequential` такая структура слоев была невозможна. Но с помощью модели `class`, теперь мы можем иметь несколько слоев, принимающих один и тот же ввод. Когда у нас есть и `conv_1x1`, и `conv_3x3`, мы объединяем их по измерению канала.

inception_module

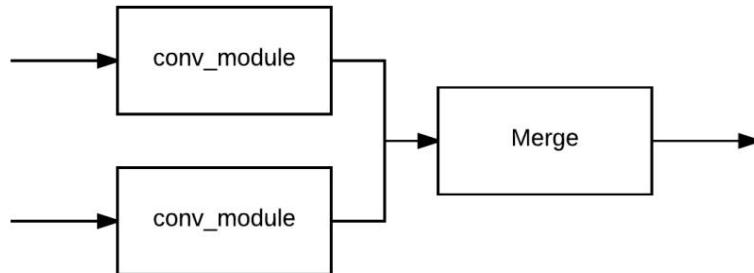


Рисунок 11.4: (мини)-`inception_module` состоит из двух ветвей. Первая ветвь - CONV слой, отвечающий за изучение фильтров 1×1 . Вторая ветвь — это еще один слой CONV, который изучает Фильтры 3×3 . Затем ответы фильтра объединяются по измерению канала.

Чтобы визуализировать модуль «Мини»-Начало, взгляните на рисунок 11.4. Наши преобразования 1×1 и 3×3 . слои принимают заданный ввод и применяют соответствующие свертки. Выход обеих сверток затем объединяется (строка 33). Нам разрешено объединять выходные данные слоя, потому что выходные данные размер тома для обеих сверток идентичен из-за `padding="same"`.

Далее следует модуль `downsample_module`, который, как следует из названия, отвечает за сокращение пространственные размеры входного объема:

```

39     @статический метод
40     def downsample_module(x, K, chanDim):

```

```

41         # определяем модуль CONV и POOL, затем объединяем
42         # по размерам канала
43
44         conv_3x3 = MiniGoogLeNet.conv_module(x, K, 3, 3, (2, 2),
45             chanDim, отступ = "действительный")
46
47         pool = MaxPooling2D ((3, 3), шаги = (2, 2)) (x)
48
49         x = объединить ([conv_3x3, pool], ось = chanDim)

50
51         # вернуть блок
52         вернуть x

```

Этот метод требует, чтобы мы передали входные данные x , количество фильтров K нашего сверточного слоя . будем учиться вместе с chanDim для пакетной нормализации и объединения каналов.

Первая ветвь модуля `downsample_module` изучает набор фильтров K , 3×3 , используя шаг 2×2 , тем самым уменьшая размер выходного тома (строки 43 и 44). Мы применяем максимальный пул на Строке 45 (вторая ветвь), снова с размером окна 3×3 и шагом 2×2 для уменьшения объема . размер. Выходные данные `conv_3x3` и `pool` затем объединяются (строка 46) и возвращаются вызывающему функция.

downsample_module

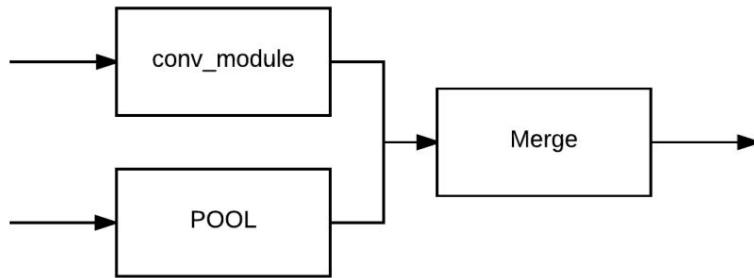


Рисунок 11.5: Модуль `downsample_module` отвечает за уменьшение пространственных размеров нашего объем ввода. Первая ветвь изучает набор фильтров с шагом 2×2 для уменьшения выходного объема. Вторая ветвь также уменьшает пространственные размеры, на этот раз применяя максимальное объединение. вывод `downsample_module` объединяется по измерению канала.

Мы можем визуализировать модуль `downsample_module` на рис. 11.5. Как видно из рисунка, кон Операции `volution` и `max pooling` применяются к одному и тому же входу, а затем объединяются.

Теперь мы готовы собрать все части вместе:

```

51     @статический метод
52
53     def build(ширина, высота, глубина, классы):
54
55         # инициализируем входную форму как "последние каналы" и
56         # сам размер каналов
57
58         inputShape = (высота, ширина, глубина)
59         ЧанДим = -1
60
61
62         # если мы используем "сначала каналы", обновите форму ввода
63         # и размер каналов
64
65         если K.image_data_format() == "channels_first":
66             inputShape = (глубина, высота, ширина)
67             ЧанДим = 1

```

Строка 52 определяет метод сборки нашей сети, что является стандартным для всех других примеров в этом примере . книга. Наш метод сборки принимает ширину, высоту, глубину и общее количество классов . что будет изучено. Строки 55 и 56 инициализируют наши `inputShape` и `chanDim` , предполагая, что мы с использованием упорядочения «каналы в последнюю очередь». Если вместо этого мы используем порядок «каналы в первую очередь», строки 60–62 обновляются . эти переменные соответственно.

Давайте определим модель Input вместе с первым conv_module:

```

64      # определяем ввод модели и первый модуль CONV
65      входы = ввод (форма = форма ввода)
66      x = MiniGoogLeNet.conv_module(входы, 96, 3, 3, (1, 1),
67      Чандим)
```

Вызов `Input` в строке 65 инициализирует архитектуру — все входы в сеть будут начинаться с этот слой, который просто «держит» входные данные (в конце концов, все сети должны иметь входные данные). первый CONV => BN => RELU применяется к строкам 66 и 67 , где мы изучаем 96 фильтров 3×3 .

Оттуда мы складываем два начальных модуля, за которыми следует модуль понижающей выборки:

```

69      # два начальных модуля, за которыми следует модуль понижающей выборки
70      x = MiniGoogLeNet.inception_module(x, 32, 32, chanDim)
71      x = MiniGoogLeNet.inception_module(x, 32, 48, chanDim)
72      x = MiniGoogLeNet.downsample_module(x, 80, chanDim)
```

Первый начальный модуль (строка 70) изучает 32 фильтра для слоев CONV 1×1 и 3×3 . При объединении этот модуль выводит объем с $K = 32 + 32 = 64$ фильтра.

Второй начальный модуль (строка 71) изучает 32 фильтра 1×1 и 48 фильтров 3×3 . Очередной раз, при объединении мы видим, что размер выходного тома составляет $K = 32 + 48 = 80$.

Модуль уменьшает размер входного тома, но сохраняет то же количество изученных фильтров, равное 80.

Далее, давайте поместим четыре начальных модуля друг на друга, прежде чем применять даунсэмплинг. позволяя GoogLeNet изучать более глубокие и богатые функции:

```

74      # четыре начальных модуля, за которыми следует модуль понижающей дискретизации
75      x = MiniGoogLeNet.inception_module(x, 112, 48, chanDim)
76      x = MiniGoogLeNet.inception_module(x, 96, 64, chanDim)
77      x = MiniGoogLeNet.inception_module(x, 80, 80, chanDim)
78      x = MiniGoogLeNet.inception_module(x, 48, 96, chanDim)
79      x = MiniGoogLeNet.downsample_module(x, 96, chanDim)
```

Обратите внимание, как в некоторых слоях мы изучаем больше фильтров 1×1 , чем фильтров 3×3 , в то время как в других модули изучают больше фильтров 3×3 , чем 1×1 . Этот тип чередующегося узора делается специально и был обоснован Szegedy et al. после проведения многих экспериментов. Когда мы реализуем более глубокую вариант GoogLeNet позже в этой главе, мы также увидим этот шаблон.

Продолжая реализацию рис. 11.2 Чжана и др., мы теперь применим еще два начальных модуля, за которыми следует глобальный пул и отсев:

```

81      # два начальных модуля, за которыми следует глобальный пул и отсев
82      x = MiniGoogLeNet.inception_module(x, 176, 160, chanDim)
83      x = MiniGoogLeNet.inception_module(x, 176, 160, chanDim)
84      x = средний пул2D ((7, 7)) (x)
85      x = отсев (0,5) (x)
```

Размер выходного тома после строки 83 составляет $7 \times 7 \times 336$. Применение среднего пула 7×7 уменьшает размер тома до $1 \times 1 \times 336$ и тем самым устраняет необходимость применения множества плотных полно связанных слоев — вместо этого мы просто среднее по пространственным выходам свертки.

Dropout применяется с вероятностью 50 процентов в строке 85, чтобы уменьшить переоснащение.

Наконец, мы добавляем наш классификатор softmax в зависимости от количества классов, которые мы хотим изучить:

```

87     # классификатор softmax x =
88     Flatten()(x) x = Dense(classes)(x)
89     = Activation("softmax")(x)
90
91
92     # создаем модель
93     модель = модель (входы, x, имя = "googlenet")
94
95     # вернуть построенную сетевую архитектуру
96     модель возврат

```

Фактическая модель затем создается в строке 93, где мы передаем входные данные, слои (x, которые включают встроенное разветвление) и, возможно, имя сети. Построенная архитектура возвращается вызывающей функции в строке 96.

11.2.2 Обучение и оценка MiniGoogLeNet на CIFAR-10

Теперь, когда MiniGoogLeNet реализован, давайте обучим его на наборе данных CIFAR-10 и посмотрим, сможем ли мы превзойти наши предыдущие лучшие 84 процента. Откройте новый файл, назовите его googlenet_cifar10.py и вставьте следующий код:

```

1 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом режиме 2 import matplotlib
matplotlib.use("Agg")
4
5 # импортируем необходимые пакеты 6 из
sklearn.preprocessing импортируем LabelBinarizer 7 из pyimagesearch.nn.conv
импортируем MiniGoogLeNet 8 из pyimagesearch.callbacks импортируем
TrainingMonitor 9 из keras.preprocessing.image импортируем ImageDataGenerator 10 из
keras.callbacks импортируем LearningRateScheduler 11 из keras.optimizers импорт SGD 12 из
keras.datasets импорт cifar10 13 импорт numpy as np 14 импорт argparse 15 импорт os

```

Строки 2 и 3 настраивают matplotlib, чтобы мы могли сохранять рисунки и графики на диск в фоновом режиме. Затем мы импортируем остальные необходимые пакеты в строках 6-15. Стока 7 импортирует нашу реализацию MiniGoogLeNet.

Также обратите внимание, как мы импортируем класс LearningRateScheduler в строке 10, что означает, что мы будем определять конкретную скорость обучения, которой должен следовать наш оптимизатор при обучении сети. В частности, мы будем определять график скорости обучения с полиномиальным затуханием. Планировщик полиномиальной скорости обучения будет следовать уравнению:

$$\alpha = \alpha_0 (1 - e/e_{\max})^p \quad (11.1)$$

Где a_0 — начальная скорость обучения, e — текущий номер эпохи, $epoch_{max}$ — максимальное число эпох, которые мы собираемся выполнить, а p — степень многочлена. Применяя это уравнение дает скорость обучения a для текущей эпохи.

Учитывая максимальное количество эпох, скорость обучения упадет до нуля. Эта скорость обучения Планировщик также можно сделать линейным, установив мощность на 1,0 — что часто и делается — и, по сути, что мы собираемся делать в этом примере. Я включил ряд примеров полиномиального обучения графики скорости с использованием максимум 70 эпох, начальной скоростью обучения $5e^{-3}$ и различной мощностью на рисунке 11.6. Обратите внимание, как с увеличением мощности скорость обучения падает быстрее. Использование силы 1,0 превращает кривую в линейный спад.

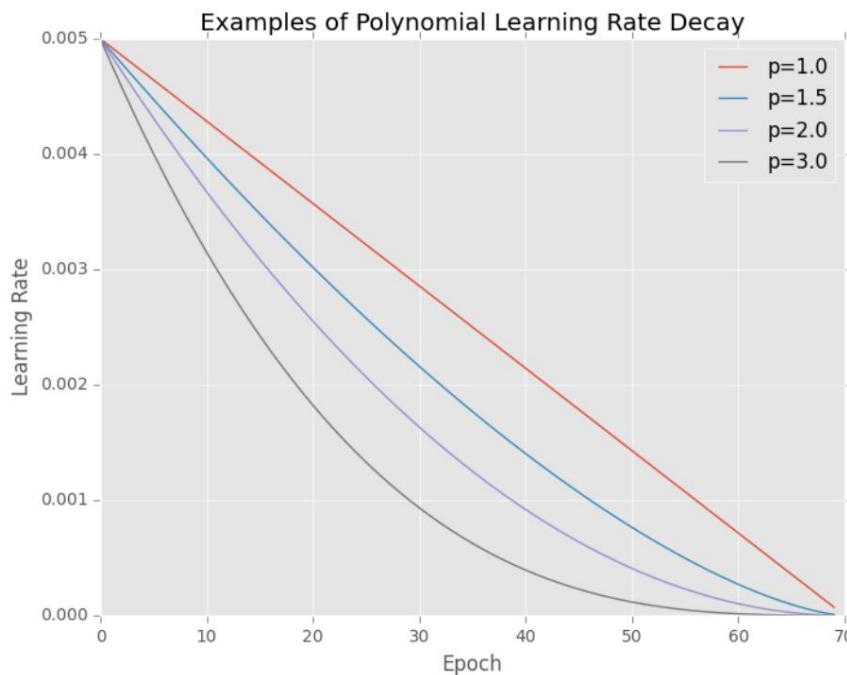


Рисунок 11.6: Графики спада полиномиальной скорости обучения для различных значений мощности, с. Уведомление как с увеличением мощности резче затухание. Установка $p = 1,0$ превращает полиномиальное затухание в линейный распад.

Давайте продолжим и реализуем эту функцию расписания скорости обучения ниже:

```

17 # определить общее количество эпох для обучения вместе с
18 # начальную скорость обучения
19 NUM_EPOCHS = 70
20 INIT_LR = 5e-3
21
22 по определению poly_decay (эпоха):
23     # инициализируем максимальное количество эпох, базовую скорость обучения,
24     # и мощность полинома
25     maxEpochs = NUM_EPOCHS
26     baseLR = INIT_LR
27     мощность = 1,0
28
29     # вычислить новую скорость обучения на основе полиномиального распада

```

```

30     альфа = baseLR * (1 - (эпоха / число с плавающей запятой (maxEpochs))) ** мощность
31
32     # вернуть новую скорость обучения
33     return alpha

```

Согласно нашему обсуждению планировщиков скорости обучения в главе 16 Starter Bundle, вы знаете, что функция планирования скорости обучения может принимать только один аргумент — текущую эпоху. Затем мы инициализируем maxEpochs, для которых разрешено обучение сети (чтобы мы могли уменьшить скорость обучения до нуля), базовую скорость обучения, а также мощность полинома.

Вычисление новой скорости обучения на основе полиномиального затухания выполняется в строке 30 — результат этого уравнения будет точно соответствовать нашим графикам на основе предоставленных параметров. Новая скорость обучения возвращается вызывающей функции в строке 33, чтобы оптимизатор мог обновить свою внутреннюю скорость обучения. Опять же, для получения дополнительной информации о планировщиках скорости обучения см. главу 16 Starter Bundle.

Теперь, когда мы определили нашу скорость обучения, мы можем проанализировать наши аргументы командной строки:

```

35 # построить разбор аргумента и разобрать аргументы 36 ap =
argparse.ArgumentParser() 37 ap.add_argument("-m", "--model", required=True,
38     help="путь к выходной модели")
39 ap.add_argument("-o", "--output", required=True,
40     help="путь к выходному каталогу (журналы, графики и т. д.)") 41
args = vars(ap.parse_args())

```

Наш скрипт требует два аргумента: --model, путь к выходному файлу, в котором находится MiniGoogLeNet. будет сериализован после обучения вместе с --output, где мы будем хранить любые графики, логи и т.д.

Следующим шагом является загрузка данных CIFAR-10 с диска, выполнение попиксельного вычитания среднего значения, а затем горячее кодирование меток:

```

43 # загрузить данные обучения и тестирования, преобразовав изображения из 44 #
целых чисел в числа с плавающей запятой 45 print("[INFO] загружает данные
CIFAR-10...") 46 ((trainX, trainY), (testX, testY)) = cifar10.load_data() 47 trainX =
trainX.astype("плавающий") 48 testX = testX.astype("плавающий")

49
50 # применить к данным вычитание среднего 51
mean = np.mean(trainX, axis=0) 52 trainX -= mean

53 testX -= среднее

54
55 # преобразовать метки из целых чисел в векторы 56 lb =
LabelBinarizer() 57 trainY = lb.fit_transform(trainY) 58 testY =
lb.transform(testY)

```

Чтобы помочь в борьбе с переоснащением и позволить нашей модели получить более высокую точность классификации, мы применить увеличение данных:

```

60 # построить генератор изображений для увеличения данных 61 aug
= ImageDataGenerator(width_shift_range=0.1,

```

```
62     height_shift_range=0.1, horizontal_flip=Истина,
63     fill_mode="близайший")
```

Мы также создадим набор обратных вызовов для отслеживания прогресса обучения, а также для вызова нашего LearningRateScheduler:

```
65 # создать набор обратных вызовов 66
figPath = os.path.sep.join([args["output"], "{}.png".format(os.getpid())])
67
68 jsonPath = os.path.sep.join([args["output"], "{}.json".format(
69     os.getpid())]) 70
обратных вызовов = [TrainingMonitor(figPath, jsonPath=jsonPath),
71     LearningRateScheduler (poly_decay)]
```

Наконец, мы готовы обучить нашу сеть:

```
73 # инициализируем оптимизатор и модель 74
print("[INFO] компилируем модель...") 75 opt =
SGD(lr=INIT_LR, импульс=0.9) 76 model =
MiniGoogLeNet.build(width=32, height=32, depth =3, классы=10) 77
model.compile(loss="categorical_crossentropy", оптимизатор=opt,
78     метрики=["точность"])
79
80 # обучаем сеть
81 print("[INFO] тренировочная сеть...") 82
model.fit_generator(aug.flow(trainX, trainY, batch_size=64),
83     validation_data=(testX, testY), steps_per_epoch=len(trainX) // 64, epochs=NUM_EPOCHS,
84     callbacks=callbacks, verbose=1)
85
86 # сохранить сеть на диск
87 print("[INFO] сериализующая сеть...") 88 model.save(args["model"])
```

Строка 75 инициализирует оптимизатор SGD с начальной скоростью обучения INIT_LR . Эта скорость обучения будет обновляться с помощью LearningRateScheduler после начала обучения. Сама архитектура MiniGoogLeNet будет принимать входные изображения шириной 32 пикселя, высотой 32 пикселя, глубиной 3 канала и всего 10 метками классов. Строки 82-84 запускают процесс обучения с использованием минипакетов размером 64, обучение в общей сложности NUM_EPOCHS . После завершения обучения Line 88 сериализует нашу модель на диск.

11.2.3 MiniGoogLeNet: эксперимент № 1 На этом

этапе работы с пакетом «Практик» для вас становится важным понять фактическое мышление, процесс и набор экспериментов, которые вам необходимо выполнить, чтобы получить высокоточную модель для заданного набора данных.

В предыдущих главах, как в этом пакете, так и в Starter Bundle, вы просто промокли ноги — и было достаточно просто увидеть фрагмент кода, понять, что он делает, выполнить его и посмотреть на результат. Это было отличной отправной точкой для развития понимания глубокого обучения.

Однако теперь, когда мы работаем с более продвинутыми архитектурами и сложными задачами, вам необходимо понять процесс проведения экспериментов, изучения результатов и последующего обновления параметров. В этой главе я предлагаю нежное введение в этот научный метод.

и следующая глава о ResNet. Если вы заинтересованы в овладении этой способностью проводить эксперименты, анализировать результаты и делать разумные постулаты относительно следующего наилучшего плана действий, обратитесь к более продвинутым урокам в ImageNet Bundle.

В моем первом эксперименте с GoogLeNet я начал с начальной скорости обучения $1e-3$ с оптимизатором SGD. Затем скорость обучения линейно уменьшалась в течение 70 эпох. Почему 70 эпох? Две причины:

1. Априорное знание. После прочтения сотен статей по глубокому обучению, сообщений в блогах, руководств и, не говоря уже о проведении собственных экспериментов, вы начнете замечать закономерность для некоторых наборов данных. В моем случае я знал из предыдущих экспериментов в своей карьере с набором данных CIFAR 10, что обычно от 50 до 100 эпох — это все, что требуется для обучения CIFAR-10. Чем глубже архитектура сети (с достаточной регуляризацией), а также снижение скорости обучения, как правило, позволяет нам обучать нашу сеть дальше. Поэтому для своего первого эксперимента я выбираю 70 эпох. После того, как эксперимент был завершен, я мог изучить график обучения и решить, следует ли использовать больше или меньше эпох (как оказалось, 70 эпох были точны).
2. Неизбежное переоснащение. Во-вторых, из предыдущих экспериментов в этой книге мы знаем, что в конечном итоге при работе с CIFAR-10 произойдет переобучение. Это неизбежно; даже при сильной регуляризации и дополнении данных это все равно произойдет. Поэтому я решил использовать 70 эпох, а не рисковать 80-100 эпохами, где эффекты переобучения станут более выраженным.

Затем я начал тренироваться, используя следующую команду:

```
$ python googlenet_cifar10.py --output output \ --model
    output/minigooglenet_cifar10.hdf5
```

Используя нашу начальную скорость обучения $1e-3$ с линейным затуханием в течение 70 эпох, мы получили точность классификации 87,95% (рис. 11.7, вверху слева). Кроме того, глядя на наш график, хотя между потерями при обучении и проверке есть разрыв, разрыв остается (относительно) пропорциональным прошлой эпохе 40. То же самое можно сказать и о кривых точности обучения и проверки. Глядя на этот график, я забеспокоился, что мы недостаточно усердно тренировались и что мы могли бы получить более высокую точность классификации со скоростью обучения $1e-2$.

11.2.4 MiniGoogLeNet: эксперимент №2

В нашем втором эксперименте MiniGoogLeNet я заменил начальную скорость обучения SGD $1e-3$ на большую $1e-2$. Эта скорость обучения линейно уменьшалась в течение 70 эпох. Я еще раз обучил сеть и собрал результаты:

```
$ python googlenet_cifar10.py --output output \ --model
    output/minigooglenet_cifar10.hdf5
```

В конце 70-й эпохи мы получаем точность 91,79 % на проверочном наборе (рис. 11.7, вверху справа), что, безусловно, лучше, чем в нашем предыдущем эксперименте, но пока не стоит слишком волноваться. Глядя на потери кривой обучения, мы видим, что они полностью падают до нуля после 60-й эпохи. Кроме того, точность классификации кривой обучения полностью насыщена на уровне 100%.

Хотя мы повысили точность проверки, мы сделали это за счет переобучения — разрыв между потерями при проверке и потерями при обучении огромен после 20-й эпохи. Вместо этого нам лучше переобучить нашу сеть с помощью обучения $5e-3$. скорость, падающий квадрат в середине $1e-2$ и $1e-3$. Мы можем получить немного меньшую точность проверки, но в идеале мы сможем уменьшить влияние переобучения.

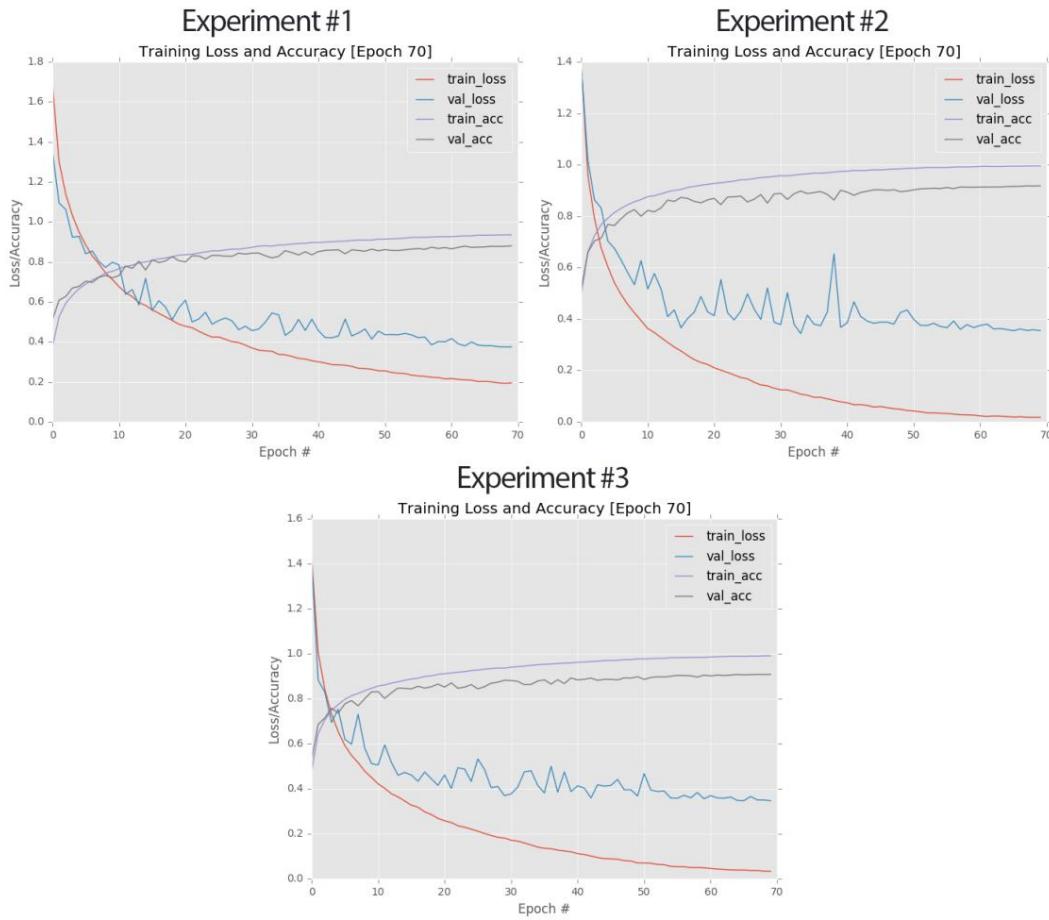


Рисунок 11.7: Вверху слева: кривые обучения для эксперимента №1. Вверху справа: Графики для эксперимента №2. Внизу: кривые обучения для эксперимента №3. Наш последний эксперимент представляет собой хороший баланс между первыми двумя, обеспечивая точность 90,81%, что выше, чем во всех наших предыдущих экспериментах CIFAR-10.

 Что бы это ни стоило, если вы обнаружите, что ваша сеть имеет полностью насыщенные потери (0,0) и точность (100%) для обучающего набора, обязательно уделите пристальное внимание своим кривым проверки. Если вы видите большие промежутки между проверочным и тренировочным графиком, вы наверняка переобулись. Вернитесь к своему эксперименту и поиграйте с параметрами, введите больше регуляризации и отрегулируйте скорость обучения. Насыщенность, подобная показанной в этом эксперименте, свидетельствует о том, что модель плохо обобщается.

11.2.5 MiniGoogLeNet: эксперимент №3

В этом эксперименте я скорректировал скорость обучения до $5e^{-3}$. MiniGoogLeNet обучался в течение 70 эпох с использованием оптимизатора SGD и линейного затухания скорости обучения:

```
$ python googlenet_cifar10.py --output output --model output/minigooglenet_cifar10.hdf5
```

Как видно из выходных данных, мы получили точность классификации 90,81% на проверочном наборе (рис. 11.7, внизу) — ниже, чем в предыдущем эксперименте, но выше, чем в первом эксперименте. Мы определенно переоснащаемся в этом эксперименте (примерно на порядок), но мы можем принять это как неизбежность при работе с CIFAR-10.

Что более важно здесь, так это то, что мы сохранили наши уровни потерь при обучении и насыщенности точности как можно ниже — потери при обучении не упали полностью до нуля, а точность обучения не достигла 100%. Мы также можем видеть разумный разрыв между точностью обучения и проверки даже в более поздние эпохи.

На данный момент я бы считал этот эксперимент первоначальным успехом (с оговоркой, что необходимо провести больше экспериментов, чтобы уменьшить переоснащение) — мы успешно обучили MiniGoogLeNet на CIFAR-10, достигнув нашей цели > 90% классификации, превзойдя все предыдущие эксперименты на CIFAR-10.

В будущих версиях этого эксперимента следует рассмотреть вопрос об агрессивной регуляризации. В частности, здесь мы не использовали какой-либо тип регуляризации веса. Применение уменьшения веса L2 помогло бы бороться с нашим переоснащением (как продемонстрируют наши эксперименты в следующем разделе).

В дальнейшем используйте этот эксперимент в качестве основы. Мы знаем, что существует переоснащение, и мы хотели бы уменьшить его, повысив точность классификации. В нашей следующей главе о ResNet мы увидим, как мы можем добиться и того, и другого.

Теперь, когда мы изучили MiniGoogLeNet применительно к CIFAR-10, давайте перейдем к более сложной задаче классификации cs231n Tiny ImageNet, где мы будем реализовывать более глубокий вариант GoogLeNet, аналогичный архитектуре, используемой Szegedy et al. в их оригиналe бумаги.

11.3 Задача Tiny ImageNet

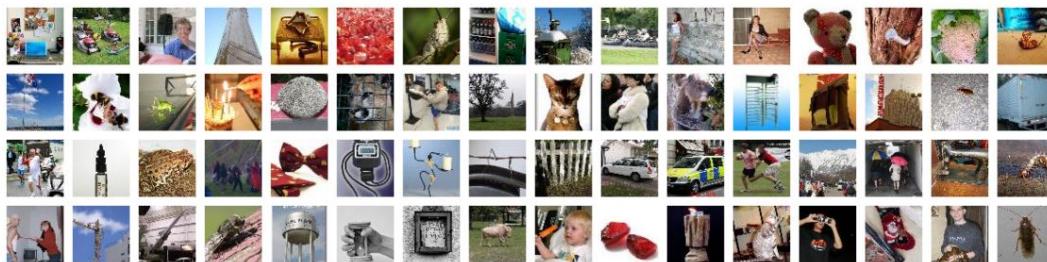


Рисунок 11.8: Образец изображений из Стенфордского конкурса классификации Tiny ImageNet.

Tiny ImageNet Visual Recognition Challenge (образец которого можно увидеть на рис . 11.8) является частью курса cs231n Стенфорда по сверточным нейронным сетям для визуального распознавания [39]. В рамках своего финального проекта учащиеся могут соревноваться в классификации, либо обучая CNN с нуля, либо выполняя трансферное обучение с помощью точной настройки (переносное обучение с помощью извлечения признаков не допускается).

Набор данных Tiny ImageNet на самом деле является подмножеством полного набора данных ImageNet (поэтому нельзя использовать извлечение признаков, поскольку это дало бы сети несправедливое преимущество), состоящего из 200 различных классов, включая все, от египетских кошек до волейбольных мячей и лимонов. Учитывая, что существует 200 классов, при случайном угадывании мы ожидаем, что будем правильными в $1/200 = 0,5\%$ случаев; поэтому наша CNN должна получить не менее 0,5%, чтобы продемонстрировать, что она изучила базовые различительные шаблоны в соответствующих классах.

Каждый класс включает 500 обучающих изображений, 50 проверочных изображений и 50 тестовых изображений. Метки наземной истины предоставляются только для обучающих и проверочных изображений. Поскольку у нас нет доступа к оценочному серверу Tiny ImageNet, мы будем использовать часть обучающего набора для формирования собственного тестового набора, чтобы мы могли оценить производительность наших алгоритмов классификации.

Как узнают читатели ImageNet Bundle (где мы обсуждаем, как обучать глубокие сверточные нейронные сети на полном наборе данных ImageNet с нуля), изображения в ImageNet

Крупномасштабные испытания визуального распознавания (ILSVRC) имеют разную ширину и высоту. Поэтому всякий раз, когда мы работаем с ILSVRC, нам сначала нужно изменить размер всех изображений в наборе данных до фиксированной ширины и высоты, прежде чем мы сможем обучить нашу сеть. Чтобы помочь учащимся сосредоточиться исключительно на компоненте глубокого обучения и классификации изображений (и не увязнуть в деталях обработки изображений), все изображения в наборе данных Tiny ImageNet были изменены до 64×64 пикселей и обрезаны по центру.

В некотором смысле изменение размера изображений делает Tiny ImageNet немного более сложным, чем его старший брат ILSVRC. В ILSVRC мы можем применять любые операции по изменению размера, обрезке и т. д., которые считаем нужными. Однако с Tiny ImageNet большая часть изображения уже отброшена для нас. Как мы узнаем, получить разумную точность ранга 1 и ранга 5 в Tiny ImageNet не так просто, как можно было бы подумать, что делает его отличным проницательным набором данных для начинающих практиков глубокого обучения, чтобы учиться и практиковаться.

В следующих нескольких разделах вы узнаете, как получить набор данных Tiny ImageNet, понять его структурировать и создавать файлы HDF5 для обучающих, проверочных и тестовых изображений.

11.3.1 Загрузка Tiny ImageNet Вы можете

загрузить набор данных Tiny ImageNet с официальной страницы списка лидеров cs231n здесь: <https://tiny-imagenet.herokuapp.com/>

В качестве альтернативы я также создал зеркало для файла здесь:

<http://pyimg.co/h28e4>. Файл .zip весит 237 МБ, поэтому убедитесь, что у вас есть подключение к Интернету, прежде чем

попытка загрузки.

11.3.2 Крошечная структура каталогов ImageNet

После загрузки и распаковки файла tiny-imagenet-200.zip разархивируйте его, и вы найдете следующую структуру каталогов:

```
--- крошечный-imagenet-200
| --- test || --- words.txt
| --- слова.txt
```

Внутри тестового каталога находятся тестовые изображения — мы будем игнорировать эти изображения, поскольку у нас нет доступа к оценочному серверу cs231n (метки намеренно не включены в загрузку, чтобы гарантировать, что никто не сможет «обмануть» в испытании).

Затем у нас есть каталог train , который содержит подкаталоги со странными именами, начинающимися с буквы n , за которой следует ряд цифр. Эти подкаталоги являются идентификаторами WordNet [43], называемыми «набором синонимов» или «синсетами» для краткости. Каждый идентификатор WordNet соответствует определенному слову/объекту. Каждое изображение внутри данного подкаталога WordNet содержит примеры этого объекта.

Мы можем найти удобочитаемую метку для идентификатора WordNet, проанализировав файл words.txt , который представляет собой просто файл, разделенный табуляцией, с идентификатором WordNet в первом столбце и удобочитаемым словом/объектом во втором столбце. В файле wnids.txt перечислены 200 идентификаторов WordNet (по одному на строку) в наборе данных ImageNet.

Наконец, в каталоге val хранится наш проверочный набор. Внутри каталога val вы найдете подкаталог images и файл с именем val_annotations.txt. val_annotations.txt предоставляет идентификаторы WordNet для каждого изображения в каталоге val.

Поэтому, прежде чем мы сможем начать обучение GoogLeNet на Tiny ImageNet, нам сначала нужно написать скрипт для разбора этих файлов и перевода их в формат HDF5. Имейте в виду, что будучи глубоким

обучение практикующему специалисту не связано с внедрением сверточных нейронных сетей и их обучением с нуля. Часть того, чтобы быть практиком глубокого обучения, включает в себя использование ваших навыков программирования для создавать простые сценарии, которые могут анализировать данные.

Чем больше у вас навыков программирования общего назначения, тем лучше вы практикуете глубокое обучение. Вы можете стать — в то время как другие исследователи глубокого обучения изо всех сил пытаются организовать файлы на диске или понять, как структурирован набор данных, вы уже преобразовали весь свой набор данных в формат, подходящий для обучения CNN.

В следующем разделе я научу вас, как определить файл конфигурации вашего проекта и создать единый, простой скрипт Python, который преобразует набор данных Tiny ImageNet в представление HDF5.

11.3.3 Создание крошечного набора данных ImageNet

Давайте продолжим и определим структуру проекта для Tiny ImageNet + GoogLeNet:

```
--- глубжегуглнет
| --- конфигурация
| | --- __init__.py
| | --- tiny_imagenet_config.py
| | --- build_tiny_imagenet.py
| | --- rank_accuracy.py
| | --- train.py
| | --- вывод/
| | | --- КПП/
| | | --- крошечное-изображение-net-200-mean.json
```

Мы создадим модуль конфигурации, в котором будем хранить любые конфигурации tiny_imagenet_config.py. Затем у нас есть скрипт build_tiny_imagenet.py, который отвечает за получение Tiny ImageNet и преобразование его в HDF5. Скрипт train.py будет обучать GoogLeNet на HDF5. версия Tiny ImageNet. Наконец, мы будем использовать rank.py для вычисления точности ранга 1 и ранга 5. для тестового набора.

Давайте продолжим и посмотрим на tiny_imagenet_config.py:

```
1 # импортируем необходимые пакеты
2 из пути импорта ОС
3
4 # определить пути к каталогам обучения и проверки
5 TRAIN_IMAGES = "../наборы данных/tiny-imagenet-200/поезд"
6 VAL_IMAGES = "../наборы данных/tiny-imagenet-200/val/images"
7
8 # определить путь к файлу, который сопоставляет имена файлов проверки с
9 # соответствующие им метки классов
10 VAL_MAPPINGS = "../наборы данных/tiny-imagenet-200/val/val_annotations.txt"
```

Строки 5 и 6 определяют пути к обучающим и проверочным изображениям Tiny ImageNet соответственно. Затем мы определяем путь к сопоставлениям файлов проверки, что позволяет нам отображать файлы проверки. имена файлов в фактические метки классов (т. е. идентификаторы WordNet).

Говоря об идентификаторах WordNet и удобочитаемых метках, давайте также определим пути к ним:

```
12 # определяем пути к используемым файлам иерархии WordNet
13 # для создания наших меток классов
14 WORDNET_IDS = "../наборы данных/tiny-imagenet-200/wnids.txt"
15 WORD_LABELS = "../datasets/tiny-imagenet-200/words.txt"
```

Учитывая, что у нас нет доступа к тестовым меткам, нам нужно взять часть обучающих данных и использовать их для проверки (поскольку у наших обучающих данных есть метки, связанные с каждым изображением):

```
17 # поскольку у нас нет доступа к данным тестирования, нам нужно 18 # взять
несколько изображений из обучающих данных и использовать их вместо этого
19 NUM_CLASSES = 200
20 NUM_TEST_IMAGES = 50 * NUM_CLASSES
```

Цель этого раздела — преобразовать Tiny ImageNet в HDF5, поэтому нам нужно указать пути к файлам обучения, проверки и тестирования HDF5:

```
22 # определить путь к выходным файлам обучения, проверки и тестирования 23 # файлы
HDF5 24 TRAIN_HDF5 = "./datasets/tiny-imagenet-200/hdf5/train.hdf5"

25 VAL_HDF5 = "./наборы данных/tiny-imagenet-200/hdf5/val.hdf5"
26 TEST_HDF5 = "./наборы данных/tiny-imagenet-200/hdf5/test.hdf5"
```

При записи изображений на диск нам нужно вычислить средние значения RGB для тренировочного набора, что позволит нам выполнить нормализацию средних значений — после того, как у нас будут средства, их нужно будет сериализовать на диск в виде файла JSON:

```
28 # определяем путь к набору данных 29 DATASET_MEAN
= "output/tiny-image-net-200-mean.json"
```

Наконец, мы определим пути к нашей выходной модели и журналам/графикам обучения:

```
31 # определить путь к выходному каталогу, используемому для хранения графиков, 32 #
отчетов о классификации и т. д.
33 OUTPUT_PATH = "выход"
34 MODEL_PATH = path.sep.join([OUTPUT_PATH,
35     "контрольные точки/epoch_70.hdf5"])
36 FIG_PATH = path.sep.join([OUTPUT_PATH,
37     "deepergooglenet_tinyimagenet.png"])
38 JSON_PATH = path.sep.join([OUTPUT_PATH,
39     "deepergooglenet_tinyimagenet.json"])
```

Как видите, этот файл конфигурации довольно прост. В основном мы просто определяем пути к входным каталогам сопоставлений изображений/меток вместе с выходными файлами. Однако время, потраченное на создание этого файла конфигурации, значительно упрощает нашу жизнь при фактическом создании Tiny ImageNet и преобразовании его в HDF5.

Чтобы понять, насколько это верно, давайте продолжим и рассмотрим `build_tiny_imagenet.py`:

```
1 # импортируем необходимые пакеты 2 из
конфига импортируем tiny_imagenet_config как конфиг 3 из
sklearn.preprocessing импортируем LabelEncoder 4 из
sklearn.model_selection импортируем train_test_split 5 из pyimagesearch.io
импортируем HDF5DatasetWriter 6 из imutils импортируем пути 7
импортируем numpy как пр
```

```
8 импорт индикатора
выполнения 9 импорт json
10 импорт cv2 11 импорт
ОС
```

Строки 2-11 импортируют необходимые пакеты Python. В строке 2 мы импортируем наш недавно закодированный файл конфигурации, чтобы иметь доступ к переменным внутри него. Мы будем использовать LabelEncoder для кодирования идентификаторов WordNet в виде целых чисел. Функция train_test_split будет применена для построения разделения обучения и тестирования. Мы будем использовать класс HDF5DatasetWriter для фактической записи необработанных изображений в соответствующие наборы данных HDF5.

Давайте продолжим и возьмем пути к тренировочным изображениям, извлечем метки классов и закодируем их:

```
13 # возьмите пути к обучающим изображениям, затем извлеките обучающие 14 # метки
классов и закодируйте их
```

```
15 trainPaths = list(paths.list_images(config.TRAIN_IMAGES)) 16 trainLabels =
[p.split(os.path.sep)[-3] для p в trainPaths] 17 le = LabelEncoder() 18 trainLabels =
le.fit_transform(метки поезда)
```

В строке 15 мы получаем список всех путей к изображениям внутри каталога TRAIN_IMAGES . Каждый путь в этот список имеет шаблон:

```
tiny-imagenet-200/train/{wordnet_id}/{уникальное_имя_файла}.JPG
```

Следовательно, чтобы извлечь идентификатор WordNet (т. е. метку класса), нам просто нужно разделить путь по разделителю пути и получить третью запись (Python имеет нулевой индекс, поэтому здесь мы указываем значение 2). Когда у нас есть все trainLabels, мы можем запустить LabelEncoder и преобразовать все метки в уникальные целые числа (строки 17 и 18).

Поскольку у нас нет тестового разделения, нам нужно выбрать набор изображений из обучающего набора, чтобы сформировать одно:

```
20 # выполнить стратифицированную выборку из обучающей выборки для построения
21 # тестовой выборки 22 split = train_test_split(trainPaths, trainLabels,
23 test_size=config.NUM_TEST_IMAGES, stratify=trainLabels, random_state=42)
24
25 (trainPaths, testPaths, trainLabels, testLabels) = разделить
```

Здесь мы указываем trainPaths и trainLabels вместе с test_size NUM_TEST_IMAGES , что составляет 50 изображений на класс (всего 10 000 изображений). Наш тестовый набор выбирается из нашего обучающего набора, где у нас уже есть метки классов для изображений, что позволяет нам оценить производительность нашей нейронной сети, когда мы будем готовы; однако мы еще не анализировали метки проверки.

Разбор и кодирование меток проверки выполняется в следующем блоке кода:

```
27 # загрузите имя файла проверки => класс из файла, а затем используйте эти
28 # сопоставления для построения путей проверки и списков меток 29 M =
open(config.VAL_MAPPINGS).read().strip().split("\n")
```

```
30 M = [r.split("\t")[:2] для r в M]
31 valPaths = [os.path.sep.join([config.VAL_IMAGES, m[0]]) для m в M]
32 valLabels = le.transform([m[1] для m в M])
```

В строке 29 мы загружаем все содержимое файла VAL_MAPPINGS (т. е. файл, разделенный табуляцией) который сопоставляет имена файлов проверочных изображений с их соответствующим идентификатором WordNet). Для каждой строки внутри M, мы разделяем его на два столбца — имя файла изображения и идентификатор WordNet (строка 30). На основе пути к проверочным изображениям (VAL_IMAGES) вместе с именами файлов в M, затем мы можем построить пути к файлам проверки (строка 31). Точно так же мы можем преобразовать строку идентификатора WordNet в уникальное целое число метки класса в строке 32 , перебирая идентификаторы WordNet в каждой строке и применяя кодировщик этикеток.

Читателям, которые изо всех сил пытаются понять этот раздел кода, я предлагаю остановиться здесь, чтобы потратить некоторое время на выполнение каждой строки и исследование содержимого каждой переменной. Мы делаем интенсивное использование списков Python здесь, которые являются естественными, лаконичными методами для построения списков с очень небольшим количеством кода. Опять же, этот блок кода не имеет ничего общего с глубоким обучением — это просто разбор файла, который является проблемой программирования общего назначения. Потратите несколько минут и убедитесь, что вы понимаете, как мы можем анализировать файл val_annotations.txt, используя этот блок кода.

Теперь, когда у нас есть пути к нашим обучающим, проверочным и тестовым изображениям, мы можем определить кортеж наборов данных, который мы будем перебирать и записывать изображения и связанные метки классов для каждого набора в HDF5 соответственно:

```
34 # составить список, объединяющий обучение, проверку и тестирование
35 # путей к изображениям вместе с соответствующими им метками и выводом HDF5
36 # файлов
37 наборов данных = [
38     ("поезд", trainPaths, trainLabels, config.TRAIN_HDF5),
39     ("val", valPaths, valLabels, config.VAL_HDF5),
40     ("тест", testPaths, testLabels, config.TEST_HDF5)]
```

Мы также инициализируем наши средние значения RGB:

```
42 # инициализируем списки усреднений каналов RGB
43 (R, G, B) = ([], [], [])
```

Наконец, мы готовы создать наши наборы данных HDF5 для Tiny ImageNet:

```
45 # цикл по кортежам набора данных
46 для (dType, paths, labels, outputPath) в наборах данных:
47     # создаем средство записи HDF5
48     print("[INFO] здание {}".format(outputPath))
49     писатель = HDF5DatasetWriter((len(paths), 64, 64, 3), outputPath)
50
51     # инициализируем индикатор выполнения
52     widgets = ["Строительный набор данных: ", progressbar.Percentage(), " ",
53               progressbar.Bar(), " ", progressbar.ETA()]
54     pbar = progressbar.ProgressBar(maxval = len(пути),
55                                   виджеты=виджеты).start()
```

В строке 46 мы перебираем тип набора данных (dType), пути, метки и outputPath в список наборов данных . Для каждого из этих выходных файлов HDF5 мы создадим HDF5DatasetWriter , который будет

хранить общее количество изображений `len(paths)`, каждое из которых представляет собой RGB-изображение $64 \times 64 \times 3$ (строка 49). Линии 52-55 просто инициализируют индикатор выполнения, чтобы мы могли легко визуализировать процесс создания набора данных.

Теперь нам нужно перебрать каждую пару путей и меток в соответствующем наборе:

```

57     # цикл по путям к изображениям
58     для (i, (путь, метка)) в перечислении (zip (пути, метки)):
59         # загружаем образ с диска
60         изображение = cv2.imread (путь)
61
62         # если мы строим обучающий набор данных, то вычисляем
63         # среднее значение каждого канала в изображении, затем обновите
64         # соответствующие списки
65         если dType == "поезд":
66             (b, g, r) = cv2.mean(изображение)[:3]
67             R.добавить(r)
68             G. добавить (g)
69             Б. добавить (б)
70
71         # добавить изображение и метку в набор данных HDF5
72         Writer.add([изображение], [метка])
73         pbar.update(я)
74
75     # закрыть модуль записи HDF5
76     pbar.finish()
77     писатель.close()
```

Для каждого изображения мы загружаем его с диска в строке 60. Если изображение является тренировочным, нам нужно вычислить среднее значение RGB изображения и обновить соответствующие списки (строки 66-69). Стока 72 добавляет изображение (которое уже имеет размер $64 \times 64 \times 3$) и метку к набору данных HDF5, в то время как строка 77 закрывает набор данных.

Последний шаг — вычислить средние значения RGB по всему набору данных и записать их на диск:

```

79 # создать словарь средних значений, затем сериализовать средние значения в
80 # JSON-файл
81 print("[INFO] сериализация означает...")
82 D = {"R": np.mean(R), "G": np.mean(G), "B": np.mean(B)}
83 f = открыть(config.DATASET_MEAN, "w")
84 f.write(json.dumps(D))
85 ф.закрыть()
```

Чтобы создать набор данных Tiny ImageNet в формате HDF5, просто выполните следующую команду:

```
$ Python build_tiny_imagenet.py
[INFO] здание ../datasets/tiny-imagenet-200/hdf5/train.hdf5...
Набор данных здания: 100% #####| Время: 0:00:36
[INFO] здание ../datasets/tiny-imagenet-200/hdf5/val.hdf5...
Набор данных здания: 100% #####| Время: 0:00:04
[INFO] здание ../datasets/tiny-imagenet-200/hdf5/test.hdf5...
Набор данных здания: 100% #####| Время: 0:00:05
[INFO] сериализация означает...
```

После завершения выполнения скрипта в каталоге hdf5 появятся три файла : train.hdf5 , val.hdf5 и test.hdf5 . Вы можете исследовать каждый из этих файлов с помощью библиотеки h5py , если хотите убедиться, что наборы данных действительно содержат изображения:

```
>>> import h5py
>>> filenames = ["train.hdf5", "val.hdf5", "test.hdf5"] >>> для имени
файла в именах файлов:
...
...      db = h5py.File(имя файла, "r")
...      print(db["images"].shape) db.close()
...
...
(90000, 64, 64, 3)
(10000, 64, 64, 3)
(10000, 64, 64, 3)
```

Мы будем использовать эти представления набора данных HDF5 для ImageNet для обучения как GoogLeNet в этой главе, так и ResNet в следующей главе.

11.4 DeeperGoogLeNet на Tiny ImageNet

Теперь, когда у нас есть HDF5-представление набора данных Tiny ImageNet, мы готовы обучить на нем GoogLeNet, но вместо использования MiniGoogLeNet, как в предыдущем разделе, мы собираемся использовать более глубокий вариант, который более точно моделирует Szegedy et al. реализация. В этом более глубоком варианте будет использоваться исходный модуль Inception, подробно описанный на рис.

11.1 ранее в этой главе, что поможет вам понять исходную архитектуру и реализовать ее самостоятельно в будущем.

Для начала мы сначала узнаем, как реализовать эту более глубокую сетевую архитектуру. Затем мы обучим DeeperGoogLeNet на наборе данных Tiny ImageNet и оценим результаты с точки зрения точности ранга 1 и ранга 5.

11.4.1 Внедрение DeeperGoogLeNet

Я предоставил рисунок (воспроизведенный и измененный из Szegedy et al.), подробно описывающий нашу архитектуру Deeper GoogLeNet на рис. 11.9. Есть только два основных различия между нашей реализацией и полной архитектурой GoogLeNet, используемой Szegedy et al. при обучении сети на полном наборе данных ImageNet:

1. Вместо использования фильтров 7×7 с шагом 2×2 в первом слое CONV мы используем фильтры 5×5 с шагом 1×1 . Мы используем их из-за того, что наша реализация GoogLeNet может принимать входные изображения только $64 \times 64 \times 3$, в то время как исходная реализация была создана для приема изображений $224 \times 224 \times 3$. Если бы мы применяли фильтры 7×7 с шагом 2×2 , мы бы слишком быстро уменьшили наши входные размеры.
2. Наша реализация немного тоньше, с двумя модулями Inception меньше — в оригинальном Szegedy et al. бумаги, еще два начальных модуля были добавлены до операции среднего пула. Этой реализации GoogLeNet будет более чем достаточно для того, чтобы мы хорошо выступили на Tiny ImageNet и заняли место в таблице лидеров cs231n Tiny ImageNet. Для читателей, которые заинтересованы в обучении полной архитектуры GoogLeNet с нуля на всем наборе данных ImageNet (таким образом воспроизводя производительность экспериментов Szegedy et al.), пожалуйста, обратитесь к главе 7 в ImageNet Bundle.

Чтобы реализовать наш класс DeeperGoogLeNet , давайте создадим файл с именем deeergooglenet.py внутри подмодуля nn.conv pyimagesearch:

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj
convolution	5×5/2	112×112×64	1						
max pool	3×3/2	56×56×64	0						
convolution	3×3/1	56×56×192	2		64	192			
max pool	3×3/2	28×28×192	0						
inception (3a)		28×28×256	2	64	96	128	16	32	32
inception (3b)		28×28×480	2	128	128	192	32	96	64
max pool	3×3/2	14×14×480	0						
inception (4a)		14×14×512	2	192	96	208	16	48	64
inception (4b)		14×14×512	2	160	112	224	24	64	64
inception (4c)		14×14×512	2	128	128	256	24	64	64
inception (4d)		14×14×528	2	112	144	288	32	64	64
inception (4e)		14×14×832	2	256	160	320	32	128	128
max pool	3×3/2	7×7×832	0						
avg pool	7×7/1	1×1×1024	0						
dropout (40%)		1×1×1024	0						
linear		1×1×1000	1						
softmax		1×1×1000	0						

Рисунок 11.9: Наша модифицированная архитектура GoogLeNet, которую мы назовем «DeeperGoogLeNet». Архитектура DeeperGoogLNet идентична исходной архитектуре GoogLeNet с двумя модификациями: (1) фильтры 5×5 с шагом 1×1 используются в первом слое CONV и (2) окончательный два начальных модуля (5a и 5b) не учитываются.

```
--- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ио
| |--- нн
| | |--- __init__.py
| | |--- конв.
| | | |--- __init__.py
| | | |--- alexnet.py
| | | |--- более глубокий googlenet.py
| | | |--- lenet.py
| | | |--- minigooglenet.py
| | | |--- minivggnet.py
| | | |--- fheadnet.py
| | | |--- мелкая сеть.py
| |--- предварительная обработка
| |--- утилиты
```

Оттуда мы можем начать работу над реализацией:

```
1 # импортируем необходимые пакеты
2 из keras.layers.normalization import BatchNormalization
3 из keras.layers.convolutional импорт Conv2D
4 из keras.layers.convolutional import AveragePooling2D
5 из keras.layers.convolutional импортировать MaxPooling2D
6 из keras.layers.core импорт Активация
7 из keras.layers.core import Dropout
```

```

8 из keras.layers.core импорт Плотный
9 из keras.layers import Flatten
10 из keras.layers import Input
11 из импорта keras.models Модель
12 из keras.layers import concatenate
13 из keras.regularizers импорт l2
14 из keras импортировать бэкэнд как K

```

Строки 2-14 начинаются с импорта необходимых пакетов Python. Обратите внимание, как мы будем использовать классы `Input` и `Model`, как в нашей реализации MiniGoogLeNet, чтобы мы могли построить граф, а не последовательная сеть — эта конструкция графа является требованием из-за того, как Ветви начального модуля. Также обратите внимание на строку 13, где мы импортируем класс `l2`, подразумевая, что мы разрешим регуляризацию веса L2 в сети, чтобы уменьшить переоснащение.

Для удобства (и чтобы наш код не раздувался) давайте определим `conv_module`, которая будет отвечать за прием входного слоя, выполнение преобразования => BN => RELU, а затем возврат вывода. Обычно я предпочитаю размещать BN после RELU, но поскольку мы копируем оригинальную работу Szegedy et al., давайте придерживаться партии нормализация перед активацией. Реализацию `conv_module` можно увидеть ниже:

```

16 класс DeeperGoogLeNet:
17     @статический метод
18     def conv_module(x, K, kX, kY, шаг, chanDim,
19                     padding="то же самое", reg=0.0005, name=None):
20         # инициализируем имена слоев CONV, BN и RELU
21         (convName, bnName, actName) = (Нет, Нет, Нет)
22
23         # если указано имя слоя, добавьте его
24         если имя не None:
25             convName = имя + "_conv"
26             bnName = имя + "_bn"
27             actName = имя + "_act"
28
29         # определить шаблон CONV => BN => RELU
30         x = Conv2D(K, (kX, kY), strides=шаг, padding=padding,
31                    kernel_regularizer=l2(reg), имя=convName)(x)
32         x = пакетная нормализация (ось = chanDim, имя = bnName)(x)
33         x = Активация ("relu", name = actName)(x)
34
35         # вернуть блок
36         вернуть x

```

Метод `conv_module` принимает ряд параметров, в том числе:

- `x`: вход в сеть.
- `K`: количество фильтров, которые будут изучать сверточный слой.
- `kX` и `kY`: размер фильтра для сверточного слоя.
- `шаг`: шаг (в пикселях) свертки. Обычно мы используем шаг 1×1 , но мы можем использовать больший шаг, если мы хотим уменьшить размер выходного тома.
- `chanDim`: это значение управляет размерностью (т.е. осью) канала изображения. Он автоматически устанавливается позже в этом классе в зависимости от того, используем ли мы «`channels_last`» или «`channels_first`» .
- `padding`: Здесь мы можем управлять заполнением слоя свертки.
- `reg`: Сила затухания веса L2.

- название: Поскольку эта сеть глубже, чем все остальные, с которыми мы работали в этой книге, мы можем хотим назвать блоки слоев, чтобы помочь нам (1) отладить сеть и (2) поделиться/объяснить сети другим.

Строка 21 инициализирует имя каждой свертки, пакетной нормализации и активации. слоев соответственно. При условии, что параметр name для conv_module не равен None, мы затем обновляем имена наших слоев (строки 24-27).

Определение шаблона слоя CONV => BN => RELU выполняется в строках 30-33 — обратите внимание как имя каждого слоя также включено. Опять же, основным преимуществом именования каждого слоя является что мы можем визуализировать имя в выводе, как мы это делали в главе 19 в Starter Bundle.

Например, использование plot_model в conv_module приведет к диаграмме, похожей на рис. 11.10.

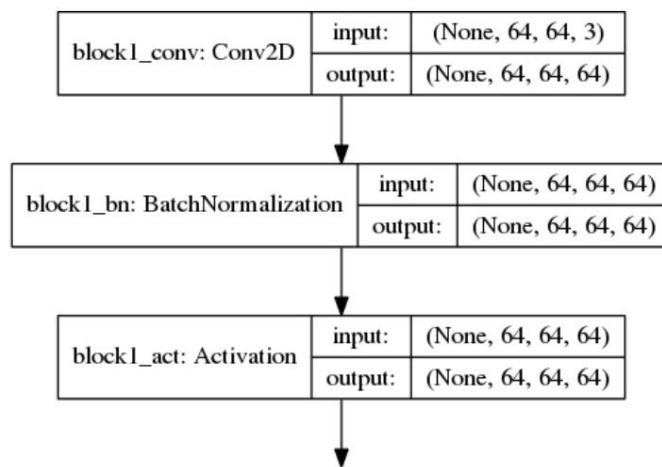


Рисунок 11.10: Пример визуализации архитектуры DeeperGoogLeNet, который включает фактические имена для каждого слоя. Именование слоев упрощает их отслеживание в более крупных и глубоких сетях.

Обратите внимание, как имена каждого слоя включены в диаграмму, что особенно полезно, когда вы работаете с глубокими CNN и можете легко «потеряться», изучая массивные диаграммы.

вывод conv_module затем возвращается вызывающей функции в строке 36.

Далее давайте определим inception_module, как подробно описано Szegedy et al. в их оригинал публикации и отображены на рис. 11.1:

```

38     @staticmethod
39     def inception_module(x, num1x1, num3x3Reduce, num3x3,
40                          num5x5Reduce, num5x5, num1x1Proj, chanDim, сцена,
41                          рег=0,0005):
42         # определяем первую ветку модуля Inception, которая
43         # состоит из сверток 1x1
44         первый = DeeperGoogLeNet.conv_module(x, num1x1, 1, 1,
45                                             (1, 1), chanDim, рег=рег, name=stage + "_first")

```

Модуль «Начало» включает в себя четыре ветви, выходы которых конкатенируются по размерность канала. Первая ветвь модуля Inception просто выполняет серию Свертки 1 × 1 — они позволяют начальному модулю изучать локальные особенности.

Вторая ветвь модуля Inception сначала выполняет уменьшение размерности через 1 × 1 свертка, за которой следует расширение свертки 3 × 3 — мы называем это нашим num3x3Reduce

и переменные num3x3 соответственно:

```

47      # определяем вторую ветку модуля Inception, которая
48      # состоит из сверток 1x1 и 3x3
49      второй = DeeperGoogLeNet.conv_module(x, num3x3Reduce, 1, 1,
50          (1, 1), chanDim, reg=reg, name=stage + "_second1")
51      второй = DeeperGoogLeNet.conv_module(второй, num3x3, 3, 3,
52          (1, 1), chanDim, reg=reg, name=stage + "_second2")

```

Здесь мы видим, что первый conv_module применяет к входным данным свертки 1×1 . Вывод этих сверток 1×1 затем передается во второй conv_module , который выполняет ряд извилин 3×3 . Количество сверток 1×1 всегда меньше числа из сверток 3×3 , тем самым выступая в качестве формы уменьшения размерности.

Третья ветка в Inception идентична второй ветке, только вместо выполнения

Сокращение 1×1 с последующим расширением 3×3 , теперь мы собираемся использовать сокращение 1×1 и расширение 5×5 :

```

54      # определяем третью ветвь модуля Inception, которая
55      # наши свертки 1x1 и 5x5
56      третий = DeeperGoogLeNet.conv_module(x, num5x5Reduce, 1, 1,
57          (1, 1), chanDim, reg=reg, name=stage + "_Third1")
58      третий = DeeperGoogLeNet.conv_module(третий, num5x5, 5, 5,
59          (1, 1), chanDim, reg=reg, name=stage + "_Third2")

```

В строках 56 и 57 мы изучаем ядра num5x5Reduce , каждое из которых имеет размер 1×1 на основе входных данных. в начальный_модуль. Затем выходные данные сверток 1×1 передаются во второй conv_module , который затем изучает фильтры num5x5 , каждый размером 5×5 . Опять же, количество Сверток 1×1 в этой ветви всегда меньше, чем количество фильтров 5×5 .

Четвертая и последняя ветвь начального модуля обычно называется пулом. проекция. Здесь мы применяем максимальное объединение, за которым следует серия сверток 1×1 :

```

61      # определяем четвертую ветвь модуля Inception, которая
62      # - проекция POOL
63      четвертый = MaxPooling2D ((3, 3), шаги = (1, 1),
64          padding="то же самое", name=стадия + "_pool")(x)
65      четвертый = DeeperGoogLeNet.conv_module(четвертый, num1x1Proj,
66          1, 1, (1, 1), chanDim, reg=reg, name=stage + "_fourth")

```

Обоснование этой ветви частично научное и частично анекдотичное. В 2014 году большинство (если не все) Сверточные нейронные сети, которые получали самую современную производительность на Набор данных ImageNet применял максимальный пул. Поэтому считалось, что CNN должна применяться максимальный пул. В то время как GoogLeNet применяет максимальный пул за пределами начального модуля, Сегеди и другие. решил включить ветку проекции пула в качестве еще одной формы максимального объединения.

Теперь, когда мы вычислили все четыре ветви, мы можем объединить их выходные данные по измерению канала и вернуть результат в вызывающую функцию:

```

68      # конкатенация по измерению канала
69      x = конкатенация ([первый, второй, третий, четвертый], ось = chanDim,
70          имя=стадия + "_mixed")
71
72      # вернуть блок
73      вернуть x

```

Если бы мы вызвали функцию `plot_model` в модуле `Inception`, используя следующую параметры:

- число 1×1 =64
- `num3x3Reduce`=96
- число 3×3 =128
- `num5x5Reduce`=16
- число 5×5 =32
- число 1×1 Proj=32

Результирующий график будет выглядеть так, как показано на рис. 11.11. На этой визуализации мы видим, как Начальный модуль строит четыре ветви. Первая ветвь отвечает за изучение местных Особенности 1×1 . Вторая ветвь выполняет уменьшение размерности с помощью свертки 1×1 , с последующим изучением фильтра большего размера 3×3 . Третья ветвь ведет себя аналогично второй ветви, изучая только фильтры 5×5 , а не фильтры 3×3 . Наконец, четвертая ветвь применяет max объединение.

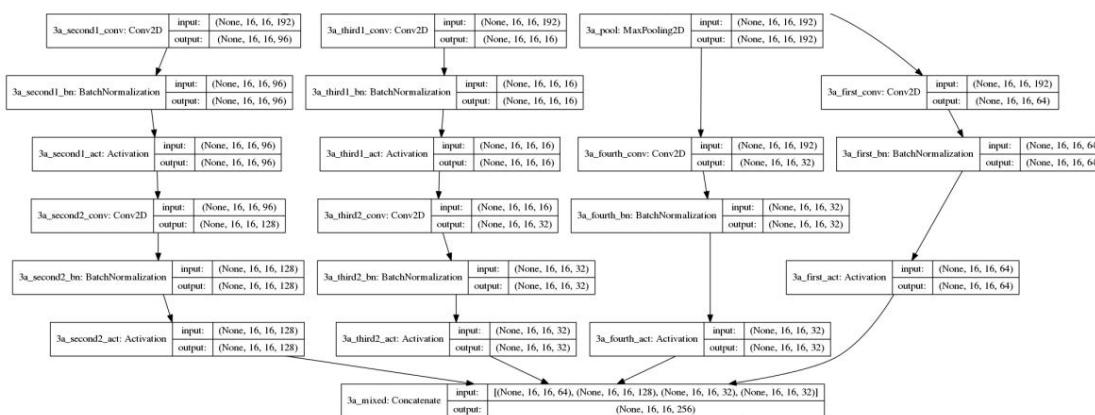


Рисунок 11.11: Полный начальный модуль, предложенный Szegedy et al. (уменьшено для экономии места). Ключевым выводом здесь является то, что в модуле `Inception` есть четыре отдельные ветви.

Изучив все три фильтра 1×1 , 3×3 и 5×5 , начальный модуль может изучить как общие, (5×5 и 3×3) вместе с локальными (1×1) элементами одновременно. Собственно процесс оптимизации будет автоматически определять, как оценивать эти ветви и слои, по сути, давая нам «общее цель», который изучит лучший набор функций (локальные, малые извилины) или более высокого уровня абстрактные функции (более крупные свертки) в данный момент времени. Таким образом, вывод начального модуля 256, что является конкатенацией всех $64+128+32+32 = 256$ фильтров из каждой ветви.

Теперь, когда `inception_module` определен, мы можем создать метод сборки , отвечающий за построение полной архитектуры DeeperGoogLeNet:

```

75     @staticmethod
76     def inception_module(width, height, depth, num_classes):
77         # инициализируем входную форму как "последние каналы" и
78         # сам размер каналов
79         inputShape = (height, width, depth)
80         ChanDim = -1
81
82         # если мы используем "сначала каналы", обновите форму ввода
83         # и размер каналов
84         if K.image_data_format() == "channels_first":
  
```

```
85         inputShape = (глубина, высота, ширина)
86         ЧанДим = 1
```

Наш метод сборки будет принимать пространственные входные размеры наших изображений, включая ширину, высоту, глубину. Мы также сможем предоставить количество меток классов, которые сеть должна изучить, вместе с необязательным членом регуляризации для уменьшения веса L2. Строки 77-86 затем обрабатываются правильно настройка inputShape и chanDim на основе конфигурации «каналы в последнюю очередь» или «каналы в первую очередь» в Керасе.

Следуя рис. 11.9 выше, наш первый блок слоев будет выполнять последовательность CONV => БАССЕЙН => (КОНВ * 2) => БАССЕЙН:

```
88     # определяем ввод модели, за которым следует последовательность CONV =>
89     # POOL => (CONV * 2) => POOL слоев
90     входы = ввод (форма = форма ввода)
91     x = DeeperGoogLeNet.conv_module(входы, 64, 5, 5, (1, 1),
92         chanDim, reg=reg, name="block1")
93     x = MaxPooling2D ((3, 3), шаги = (2, 2), отступ = "то же",
94         имя="пул1")(x)
95     x = DeeperGoogLeNet.conv_module(x, 64, 1, 1, (1, 1),
96         chanDim, reg=reg, name="block2")
97     x = DeeperGoogLeNet.conv_module(x, 192, 3, 3, (1, 1),
98         chanDim, reg=reg, name="block3")
99     x = MaxPooling2D ((3, 3), шаги = (2, 2), отступ = "то же",
100        имя="пул2")(x)
```

Первый слой CONV изучает 64 фильтра 5×5 с шагом 1×1 . Затем мы применяем максимальный пул с размером окна 3×3 и шагом 2×2 , чтобы уменьшить размер объема ввода. Линии 95-98 отвечают за выполнение сокращения и расширения. Сначала изучаются 64 фильтра 1×1 (строки 95 и 96). Затем в строках 97 и 98 изучаются 192 фильтра 3×3 . Этот процесс очень похож на начальном модулю (который будет применяться на более поздних уровнях сети), только без фактора ветвления. Наконец, еще одно максимальное объединение выполняется в строках 99 и 100.

Затем давайте применим два начальных модуля (За и 3b), а затем максимальное объединение:

```
102    # применить два начальных модуля, за которыми следует пул
103    x = DeeperGoogLeNet.inception_module(x, 64, 96, 128, 16,
104        32, 32, чандим, "За", обр=рег)
105    x = DeeperGoogLeNet.inception_module(x, 128, 128, 192, 32,
106        96, 64, чандим, "3б", обр=рег)
107    x = MaxPooling2D ((3, 3), шаги = (2, 2), отступ = "то же",
108        имя="пул3")(x)
```

Глядя на этот код, вы можете задаться вопросом, как мы определили количество фильтров для каждого CONV. слой. Ответ заключается в том, что все значения параметров в этой сети были взяты непосредственно из исходного Сегеди и др. статья на GoogLeNet, где авторы провели ряд экспериментов по настройке параметры. В любом случае вы заметите общий шаблон для всех модулей Inception:

1. Количество фильтров 1×1 , которые мы изучаем в первой ветке начального модуля, будет меньше больше или равно фильтрам 1×1 в ветвях 3×3 (вторая) и 5×5 (третья).
2. Количество фильтров 1×1 всегда будет меньше сверток 3×3 и 5×5 , которые они кормить в.
3. Количество фильтров, которые мы изучим в ветке 3×3 , будет больше, чем в ветке 5×5 , которая помогает уменьшить размер сети, а также повышает скорость обучения/оценки.

4. Количество проекционных фильтров пула всегда будет меньше первой ветки 1×1 локального Особенности.

5. Независимо от типа ветки, количество фильтров будет увеличиваться (или, по крайней мере, останется прежним) по мере углубления в сеть.

Хотя, безусловно, есть и другие параметры, которые необходимо отслеживать при реализации этой сети, мы по-прежнему следуем тем же общим практическим правилам, что и в предыдущих CNN — чем глубже сеть получает, тем меньше размер тома; следовательно, чем больше фильтров мы учимся компенсировать.

Сеть продолжает становиться глубже, изучая более богатые функции, поскольку теперь мы складываем пять начальных модули (4a-4e) друг над другом перед применением POOL:

```

110      # применить пять начальных модулей, за которыми следует POOL
111      x = DeeperGoogLeNet.inception_module(x, 192, 96, 208, 16,
112          48, 64, чандим, "4a", обр=рег)
113      x = DeeperGoogLeNet.inception_module(x, 160, 112, 224, 24,
114          64, 64, чандим, "4b", обр=рег)
115      x = DeeperGoogLeNet.inception_module(x, 128, 128, 256, 24,
116          64, 64, чандим, "4c", обр=рег)
117      x = DeeperGoogLeNet.inception_module(x, 112, 144, 288, 32,
118          64, 64, чандим, "4d", обр=рег)
119      x = DeeperGoogLeNet.inception_module(x, 256, 160, 320, 32,
120          128, 128, chanDim, "4e", reg=reg)
121      x = MaxPooling2D ((3, 3), шаги = (2, 2), отступ = "то же",
122          имя="пул4")(x)

```

После финального POOL на линиях 121 и 122 размер нашего объема составляет $4 \times 4 \times$ класса. Чтобы избежать использование дорогостоящих в вычислительном отношении полно связанных слоев (не говоря уже о резком увеличении размер сети), мы применяем средний пул с ядром 4×4 , чтобы уменьшить размер тома до $1 \times 1 \times$ классы:

```

124      # применить слой POOL (средний) с последующим удалением
125      x = средний пул2D((4, 4), name="pool5")(x)
126      x = отсев (0,4, name="do")(x)
127
128      # классификатор softmax
129      x = сгладить (имя = "сгладить") (x)
130      x = плотный (классы, kernel_regularizer = l2 (reg),
131          имя="метки")(x)
132      x = Активация ("softmax", name="softmax")(x)
133
134      # создаем модель
135      модель = модель (входы, x, имя = "googlenet")
136
137      # вернуть построенную сетевую архитектуру
138      модель возврата

```

Затем применяется отсев с вероятностью 40%. Обычно мы используем коэффициент отсева 50%. но опять же, мы просто следуем исходной реализации.

Строки 130 и 131 создают плотный слой для общего количества классов, которые мы хотим изучить. Затем после полно связного слоя в строке 132 применяется классификатор softmax. Модель строится на основе входных данных и x, фактического графа вычислительной сети. Этот Модель возвращается вызывающей функции в строке 138.

11.4.2 Обучение DeeperGoogLeNet на Tiny ImageNet

Теперь, когда наша архитектура DeeperGoogLeNet реализована, нам нужно создать скрипт Python, который будет обучать сеть на Tiny ImageNet. Нам также потребуется создать второй скрипт Python, который будет отвечать за оценку нашей модели на тестовом наборе путем вычисления точности ранга 1 и ранга 5.

Как только мы выполним обе эти задачи, я поделюсь тремя экспериментами, которые я провел, собирая результаты для этой главы. Эти эксперименты сформируют «кейс» и позволят вам узнать, как проводить эксперимент, исследовать результаты и сделать обоснованное предположение о том, как настроить ваши гиперпараметры, чтобы получить более производительную сеть в вашем следующем эксперименте.

11.4.3 Создание сценария обучения

Давайте продолжим и реализуем сценарий обучения — откройте новый файл, назовите его train.py и вставьте следующий код:

```

1 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом режиме 2
import matplotlib
matplotlib.use("Agg")

4
5 # импортируем необходимые пакеты 6
из config import tiny_imagenet_config как config 7 из
pyimagesearch.preprocessing import ImageToArrayPreprocessor 8 из
pyimagesearch.preprocessing import SimplePreprocessor 9 из pyimagesearch.preprocessing
import MeanPreprocessor 10 из pyimagesearch.callbacks import EpochCheckpoint 11 из
pyimagesearch.callbacks1 import from Training2Monitor pyimagesearch.io импортировать
HDF5DatasetGenerator 13 из pyimagesearch.nn.conv импортировать DeeperGoogLeNet
14 из keras.preprocessing.image импортировать ImageDataGenerator 15 из
keras.optimizers импортировать Adam 16 из keras.models импортировать load_model
17 импортировать keras.backend as K 18 импортировать argparse 19 импортировать
json

```

Строки 2 и 3 указывают библиотеке matplotlib использовать серверную часть, чтобы мы могли сохранять графики потерь и точности на диск. Затем мы импортируем остальные необходимые пакеты Python в строках 6-19. Взгляните на строку 6, где мы импортируем файл конфигурации для эксперимента Tiny ImageNet. Мы также импортируем нашу реализацию DeeperGoogLeNet в строку 13. Теперь все операции импорта должны показаться вам относительно знакомыми.

Оттуда мы можем проанализировать наши аргументы командной строки:

```

21 # построить разбор аргумента и разобрать аргументы 22 ap =
argparse.ArgumentParser() 23 ap.add_argument("-c", "--checkpoints",
required=True,
24         help="путь к выходному каталогу контрольных точек")
25 ap.add_argument("-m", "--model", type=str, help="путь к
26         *конкретной* контрольной точке модели для загрузки") 27
ap.add_argument("-s", "--start-epoch ", тип=целое, по умолчанию=0,
28         help="эпоха для возобновления обучения")
29 args = vars(ap.parse_args())

```

Мы будем использовать метод `ctrl + c` для обучения нашей сети, что означает, что мы начнем процесс обучения, отслеживаем, как идет обучение, а затем остановим скрипт, если произойдет переобучение/застой, отрегулируйте любые гиперпараметры и перезапустите обучение. Для начала нам понадобится `--checkpoints switch`, который представляет собой путь к выходному каталогу, в котором будут храниться отдельные контрольные точки для Модель DeeperGoogLeNet . Если мы перезапускаем обучение, нам нужно указать путь к конкретная `--model` , с которой мы перезапускаем обучение. Точно так же нам также необходимо предоставить `--start-epoch` для получения целочисленного значения эпохи, с которой мы перезапускаем обучение.

Чтобы получить разумную точность в наборе данных Tiny ImageNet, нам нужно применить данные дополнение к тренировочным данным:

```
31 # построить генератор обучающих изображений для увеличения данных
32 августа = ImageDataGenerator (rotation_range = 18, zoom_range = 0,15,
33     width_shift_range=0,2, height_shift_range=0,2, shear_range=0,15,
34     horizontal_flip = Истина, fill_mode = "ближайший")
35
36 # загрузить средства RGB для тренировочного набора
37 означает = json.loads(open(config.DATASET_MEAN).read())
```

Мы также загрузим наши средства RGB (строка 37) для вычитания среднего и нормализации. Давайте перейдем к созданию экземпляров обоих наших препроцессоров изображений, а также обучения и Генераторы набора данных проверки HDF5:

```
39 # инициализируем препроцессоры изображений
40 sp = Простой препроцессор (64, 64)
41 mp = MeanPreprocessor(означает ["R"], означает ["G"], означает ["B"])
42 iap = ImageToArrayPreprocessor()
43
44 # инициализировать генераторы наборов данных для обучения и проверки
45 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 64, авг = авг,
46     препроцессоры=[sp, mp, iap], классы=config.NUM_CLASSES)
47 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
48     препроцессоры=[sp, mp, iap], классы=config.NUM_CLASSES)
```

Оба генератора обучения и проверки будут применяться:

1. Простой препроцессор, обеспечивающий изменение размера изображения до 64×64 пикселей (что уже должен быть, но мы включим его сюда для полноты картины).
2. Вычитание среднего для нормализации данных.
3. Преобразователь изображения в Keras-совместимый массив.

Мы будем обучать нашу сеть в мини-пакетах размером 64. В случае, если мы обучаем DeeperGoogLeNet с первой эпохи, `wet` должен создать сеть и оптимизатор:

```
50 # если контрольная точка конкретной модели не указана, инициализировать
51 # сеть и компилируем модель
52 , если args["model"] равно None:
53     print("[INFO] компилируемая модель...")
54     модель = DeeperGoogLeNet.build (ширина = 64, высота = 64, глубина = 3,
55         классы = config.NUM_CLASSES, регистр = 0,0002)
56     опт = Адам (1e-3)
57     model.compile(потеря="categorical_crossentropy", оптимизатор=опт,
58         метрики=["точность"])
```

Обратите внимание, как мы применяем силу регуляризации L2, равную 0,0002 , а также метод Адама. оптимизатор – мы узнаем почему в Разделе 11.4.5 ниже.

В противном случае мы должны перезапустить обучение с определенной эпохи, поэтому нам нужно загрузить модель. и отрегулируйте скорость обучения:

```

60 # иначе загрузить чекпойнт с диска
61 еще:
62     print("[INFO] загрузка {}".format(args["model"]))
63     модель = load_model(аргументы ["модель"])
64
65     # обновить скорость обучения
66     print("[INFO] старая скорость обучения: {}".format(
67         K.get_value(model.optimizer.lr)))
68     K.set_value(model.optimizer.lr, 1e-5)
69     print("[INFO] новая скорость обучения: {}".format(
70         K.get_value(model.optimizer.lr)))

```

Мы создадим два обратных вызова, один для сериализации весов модели на диск каждые пять эпох и другой, чтобы построить наш график потерь/точности с течением времени:

```

72 # построить набор обратных вызовов
73 обратных вызова =
74     EpochCheckpoint(args["контрольные точки"], каждый=5,
75                     startAt=args["start_epoch"]),
76     TrainingMonitor (config.FIG_PATH, jsonPath = config.JSON_PATH,
77                     startAt=args["start_epoch"]))

```

Наконец, мы можем обучить нашу сеть:

```

79 # обучаем сеть
80 модель.fit_generator(
81     поездГен.генератор(),
82     steps_per_epoch=trainGen.numImages // 64,
83     validation_data=valGen.generator(),
84     validation_steps=valGen.numImages // 64,
85     эпохи=10,
86     max_queue_size=64 * 2,
87     обратные вызовы = обратные вызовы, подробный = 1)
88
89 # закрыть базы
90 поездГен.закрыть()
91 valGen.close()

```

Точное количество эпох, которые мы выбираем для обучения нашей сети, будет зависеть от того, как графики потерь/точности. Мы примем обоснованное решение относительно обновления показателей обучения или применение ранней остановки на основе производительности модели.

11.4.4 Создание сценария оценки

Как только мы будем удовлетворены производительностью нашей модели на обучающем и проверочном наборе, мы можем перейти к оценке сети на тестовом наборе. Для этого создадим новый файл с именем rank_accuracy.py:

```

1 # импортируем необходимые пакеты 2
из config import tiny_imagenet_config as config 3 из
pyimagesearch.preprocessing import ImageToArrayPreprocessor 4 из
pyimagesearch.preprocessing import SimplePreprocessor 5 из pyimagesearch.preprocessing
import MeanPreprocessor 6 из pyimagesearch.utils.ranked import rank5_accuracy 7 из
pyimagesearch.io import HDF5DatasetGenerator 8 из keras.models импортировать
load_model 9 импортировать json

```

Для начала мы импортируем необходимые пакеты Python. Функция rank5_accuracy импортирована в строке 6 , чтобы мы могли вычислить точность ранга 1 и ранга 5 в наборе данных.

Оттуда мы загружаем наши средства RGB, инициализируем наши препроцессоры изображений (так же, как мы делали это для тестирования), а затем инициализируем генератор тестового набора данных:

```

11 # загрузить средства RGB для тренировочного
набора 12 mean = json.loads(open(config.DATASET_MEAN).read())
13
14 # инициализируем препроцессоры
изображений 15 sp = SimplePreprocessor(64, 64)
16 mp = MeanPreprocessor(means["R"], означает["G"], означает["B"]) 17 iap =
ImageToArrayPreprocessor()
18
19 # инициализируем генератор набора тестовых данных
20 testGen = HDF5DatasetGenerator(config.TEST_HDF5, 64,
препроцессоры=[sp, mp, iap], классы=config.NUM_CLASSES)

```

Следующий блок кода обрабатывает загрузку предварительно обученной модели с диска через MODEL_PATH, которую мы указали в нашем файле конфигурации:

```

23 # загрузить предварительно обученную
сеть 24 print("[INFO] loading model...") 25
model = load_model(config.MODEL_PATH)

```

Вы должны установить MODEL_PATH в качестве последней контрольной точки эпохи после завершения обучения. В качестве альтернативы вы можете установить эту переменную на более ранние эпохи, чтобы получить представление о том, как повышается точность тестирования в более поздние эпохи.

Как только модель загружена, мы можем делать прогнозы по данным тестирования и отображать как точность ранга 1 и ранга 5:

```

27 # делать прогнозы на основе тестовых данных
28 print("[INFO] прогнозирование на тестовых данных...")
29 прогнозы = model.predict_generator(testGen.generator(),
30 steps=testGen.numImages // 64, max_queue_size=64 * 2)
31
32 # вычислить точность rank-1 и rank-5 33 (rank1, rank5)
= rank5_accuracy(predictions, testGen.db["labels"]) 34 print("[INFO] rank-1: {:.2f}%
%" .format(rank1 * 100)) 35 print("[INFO] rank-5: {:.2f}%".format(rank5 * 100))
36
37 # закрыть базу
38 testGen.close()

```

Скорость обучения эпохи 1-	
25	1e-2
26	35 1e-3
36	65 1e-
4	

Таблица 11.1: График скорости обучения, использованный при обучении DeeperGoogLeNet на Tiny ImageNet для эксперимента №1.

11.4.5 Эксперименты DeeperGoogLeNet

В следующих разделах я включил результаты четырех отдельных экспериментов, которые я проводил при обучении DeeperGoogLeNet на Tiny ImageNet. После каждого эксперимента я оценивал результаты, а затем принимал взвешенное решение о том, как следует обновить гиперпараметры и сетевую архитектуру, чтобы повысить точность.

Подобные тематические исследования особенно полезны для вас, как для начинающего специалиста по глубокому обучению. Они не только демонстрируют, что глубокое обучение — это итеративный процесс, требующий множества экспериментов, но также показывают, на какие параметры следует обращать внимание и как их обновлять.

Наконец, стоит отметить, что некоторые из этих экспериментов требовали внесения изменений в код. Обе реализации deepgooglenet.py и train.py являются моими последними реализациями, получившими наилучшую точность. Я отмечу изменения, внесенные в более ранние эксперименты, на случай, если вы захотите воспроизвести мои (менее точные) результаты.

DeeperGoogLeNet: эксперимент №1 Учитывая, что

я впервые тренировал сеть на задаче Tiny ImageNet, я не был уверен, какой должна быть оптимальная глубина для данной архитектуры в этом наборе данных. Хотя я знал, что Tiny ImageNet будет сложной задачей классификации, я не думал, что начальные модули 4a-4e потребуются, поэтому я удалил их из нашей реализации DeeperGoogLeNet выше, что привело к значительно более поверхностной сетевой архитектуре.

Я решил обучить DeeperGoogLeNet с использованием SGD с начальной скоростью обучения $1e - 2$ и импульсом 0,9 (ускорение Нестерова не применялось). Я всегда использую SGD в своем первом эксперименте. Согласно моим рекомендациям и эмпирическим правилам в главе 7, вы должны сначала попробовать SGD, чтобы получить базовый уровень, а затем, если необходимо, использовать более продвинутые методы оптимизации.

Я начал обучение, используя следующую команду:

```
$ python train.py --вывод контрольных точек/контрольные точки
```

Затем был использован график скорости обучения, подробно описанный в таблице 11.1. Из этой таблицы следует, что после 25-й эпохи я прекратил обучение, снизил скорость обучения до $1e - 3$, а затем возобновил обучение еще на 10 эпох:

```
$ python train.py --checkpoints вывод/контрольные точки \
--model output/checkpoints/epoch_25.hdf5 --start-epoch 25
```

После эпохи 35 я снова прекратил обучение, снизил скорость обучения до $1e - 4$, а затем возобновил тренировку еще на тридцать эпох:

```
$ python train.py --checkpoints вывод/контрольные точки \
--model output/checkpoints/epoch_35.hdf5 --start-epoch 35
```

Обучение в течение дополнительных тридцати эпох было, мягко говоря, чрезмерным; тем не менее, я хотел почувствовать уровень переобучения, который можно ожидать для большого количества эпох после того, как исходная скорость обучения была снижена (поскольку я впервые работал с GoogLeNet + Tiny ImageNet). На рис. 11.12 (вверху слева) вы можете увидеть график потерь/точности с течением времени как для обучения, так и для проверки .

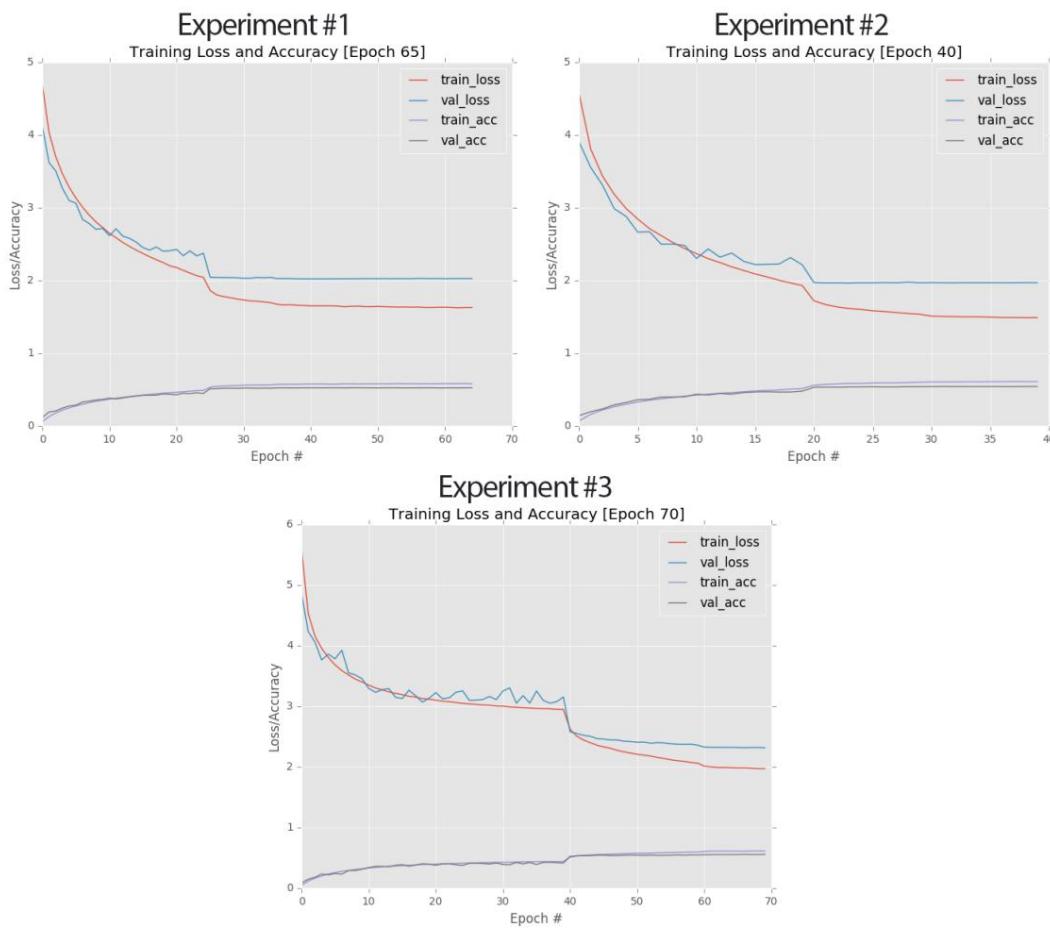


Рисунок 11.12: Вверху слева: Графики для эксперимента №1. Вверху справа: кривые обучения для эксперимента №2. Внизу: графики обучения/проверки для эксперимента №3. Последний эксперимент дает наилучшую точность проверки на уровне 55,77%.

Начиная примерно с 15-й эпохи, наблюдается расхождение в обучении и потерях проверки. К тому времени, когда мы добираемся до эпохи 25, расхождение становится более значительным, поэтому я снизил скорость обучения на порядок; результатом является хороший скачок в точности и снижение потерь. Проблема в том , что после этого момента как обучение, так и проверочное обучение практически стагнируют. Даже снижение скорости обучения до $1e-4$ в эпоху 35 не дает дополнительного повышения точности.

В конце эпохи 40-х обучение полностью застопорилось. Если бы я не хотел видеть последствия низкой скорости обучения в течение длительного периода времени, я бы прекратил обучение после 45-й эпохи. В этом случае я позволил сети обучаться до 65-й эпохи (без изменения потерь/точности) . где я остановил обучение и изучил результаты, отметив, что сеть достигла 52,25% точности ранга 1 на проверочном наборе. Однако, учитывая, как производительность нашей сети быстро стабилизировалась после снижения скорости обучения, я решил, что явно предстоит проделать еще много работы.

Скорость обучения эпохи		Скорость обучения эпохи			
1	20 1e	3	1	40 1e	3
21	30 1e	4	41	60 1e	4
31	40 1e	5	61	70 1e	5

Таблица 11.2. Слева: график скорости обучения, использованный при обучении DeeperGoogLeNet на Tiny ImageNet в эксперименте № 2. Справа: график скорости обучения для эксперимента № 3.

DeeperGoogLeNet: эксперимент №2 В

моем втором эксперименте с DeeperGoogLeNet + Tiny ImageNet я решил отключить оптимизатор SGD для Адама. Это решение было принято исключительно потому, что я не был убежден, что сетевая архитектура должна быть более глубокой (пока). Оптимизатор Adam использовался с начальной скоростью обучения по умолчанию 1×10^{-3} . Затем я использовал график скорости обучения в таблице 11.2 (слева), чтобы снизить скорость обучения.

График кривых обучения можно увидеть ниже на рис. 11.12 (вверху справа). Этот график очень похож на верхний левый вверху. Сначала мы начинаем хорошо, но потери при проверке быстро расходятся после эпохи 10, что вынуждает меня снизить скорость обучения в эпоху 20 (или иначе рисковать переоснащением). Как только скорость обучения снижается, обучение выходит на плато, и я не могу повысить точность, даже снижая скорость обучения во второй раз. Однако в конце 40-й эпохи я заметил, что мои потери при валидации были ниже, чем в предыдущем эксперименте, а моя точность была выше.

Заменив SGD на Адама, я смог повысить точность проверки до 54,20% Rank -1, то есть почти на 2%. Тем не менее, у меня все еще была проблема стагнации обучения, как только начальная скорость обучения была снижена.

DeeperGoogLeNet: эксперимент № 3

Учитывая стагнацию обучения, я постулировал, что сеть недостаточно глубокая, чтобы смоделировать базовые шаблоны в наборе данных Tiny ImageNet. Поэтому я решил включить начальные модули 4a-4e, создав гораздо более глубокую сетевую архитектуру, способную изучать более глубокие и отличительные функции. Для обучения сети использовался оптимизатор Adam с начальной скоростью обучения 1×10^{-3} . Я оставил коэффициент затухания веса L2 равным 0,0002. Затем DeeperGoogLeNet был обучен в соответствии с таблицей 11.2 (справа).

Вы можете увидеть график на рис. 11.12 (внизу). Вы сразу заметите, что использование более глубокой сетевой архитектуры позволило мне тренироваться дольше, не рискуя стагнацией или серьезным переоснащением. В конце 70-й эпохи я получал 55,77% точности ранга 1 на проверочном наборе.

В этот момент я решил, что пришло время оценить DeeperGoogLeNet на тестовом наборе:

```
$ python rank_accuracy.py [INFO]
модель загружена...
[INFO] прогнозирование на основе тестовых данных...
[ИНФО] ранг-1: 54,38%
[ИНФО] ранг-5: 78,96%
```

Сценарий оценки сообщил о точности ранга 1, равной 54,38%, или частоте ошибок 1 – 0,5438 = 0,4562. Также интересно отметить, что наша точность 5-го ранга составляет 78,96%, что весьма впечатляет для этой задачи. Глядя на таблицу лидеров Tiny ImageNet (<http://pyimg.co/h5q0o>) ниже мы видим, что этой частоты ошибок достаточно, чтобы претендовать на позицию № 7, отличное начало пути к вершине таблицы лидеров (рис. 11.13).

Leaderboard

#	Name	Error Rate	# Submissions
1	Avati,Anand	0.268	14
2	Kim,Hansohl Elliott	0.311	17
3	Qian,Junyang	0.338	6
4	Liu,Fei	0.339	8
5	Zhai,Andrew Huan	0.446	4
6	Shen,William	0.452	9
7	Shcherbina,Anna	0.506	15
8	Ebrahimi,Mohammad Sadegh	0.561	5
9	Ting,Jason Ming	0.616	17
10	Random Guess	0.995	17
11	Khosla,Vani	0.995	4

Рисунок 11.13: Наша первая успешная попытка обучить DeeperGoogLeNet на наборе данных Tiny ImageNet позволила нам занять 7-е место в таблице лидеров. Мы сможем достичь более высоких позиций в нашей следующей главе о ResNet.

 Позиции 1-4 в таблице лидеров Tiny ImageNet были достигнуты за счет передачи обучения путем точной настройки сетей, уже обученных на полном наборе данных ImageNet. Поскольку мы обучаем наши сети с нуля, мы больше озабочены тем, чтобы претендовать на позицию № 5, самую высокую позицию, достигнутую без трансферного обучения. Как мы узнаем из следующей главы о ResNet, мы легко сможем претендовать на эту позицию. Для получения дополнительной информации о методах, которые студенты cs231n использовали для достижения своей частоты ошибок, см. Страницу проекта Stanford cs231n [4].

Для читателей, заинтересованных в дальнейшем повышении точности DeeperGoogLeNet, я бы предложил следующие эксперименты: 1. Измените conv_module , чтобы он использовал CONV => RELU => BN вместо оригинального CONV => BN .
 => Заказ RELU.
 2. Попытка использовать ELU вместо ReLU, что, вероятно, приведет к небольшому приросту на 0,5-1% в точность.

11.5 Резюме

В этой главе мы рассмотрели работу Szegedy et al. [17], которая представила ныне известный модуль Inception. Начальный модуль является примером микроархитектуры, строительным блоком , который вписывается в общую макроархитектуру сети. Текущее состояние Нейронные сети, как правило, используют некоторую форму микроархитектуры.

Затем мы применили модуль Inception для создания двух вариантов GoogLeNet: 1. Один для CIFAR-10.

2. И еще один для более сложной Tiny ImageNet.

При обучении на CIFAR-10 мы добились нашей наилучшей точности 90,81% (и улучшение по сравнению с предыдущим лучшим показателем 84%).

В сложном наборе данных Tiny ImageNet мы достигли 54,38% точности ранга 1 и 78,96% ранга 5 на тестовом наборе, что позволило нам занять 7-е место в таблице лидеров Tiny ImageNet.

Наша цель — подняться в таблице лидеров на позицию № 5 (самая высокая позиция, полученная при обучении сети с нуля, все более высокие позиции применяли тонкую настройку на сетях, предварительно обученных на

набор данных ImageNet, что дает им несправедливое преимущество). Чтобы достичь нашей цели позиции № 5, нам нужно используйте архитектуру ResNet, подробно описанную в следующей главе.

12. Ренет

В нашей предыдущей главе мы обсудили архитектуру GoogLeNet и модуль Inception, микроархитектуру, которая действует как строительный блок в общей макроархитектуре. Теперь мы собираемся обсудить другую сетевую архитектуру, основанную на микроархитектурах — ResNet.

ResNet использует так называемый остаточный модуль для обучения сверточных нейронных сетей на глубинах, которые ранее считались невозможными. Например, в 2014 году очень глубокими считались архитектуры VGG16 и VGG19 [11]. Однако с помощью ResNet мы успешно обучили сети с более чем 100 слоями в сложном наборе данных ImageNet и более 1000 слоев в CIFAR-10 [24].

Такая глубина стала возможной только благодаря использованию «умных» алгоритмов инициализации весов (таких как Xavier/Glorot [44] и MSRA/He и др. [45]) вместе с тождественным отображением — концепцией, которую мы обсудим позже в этой главе. Учитывая глубину сетей ResNet, возможно, неудивительно, что ResNet заняла первое место во всех трех задачах ILSVRC 2015 (классификация, обнаружение и локализация).

В этой главе мы собираемся обсудить архитектуру ResNet, остаточный модуль, а также обновления остаточного модуля, которые сделали его способным обеспечить более высокую точность классификации. Оттуда мы внедрим и обучим варианты ResNet на наборе данных CIFAR-10 и вызове Tiny ImageNet — в каждом случае наши реализации ResNet превзойдут все эксперименты, которые мы проводили в этой книге.

12.1 ResNet и остаточный модуль

Впервые представлено He et al. в их статье 2015 года «Глубокое остаточное обучение для распознавания изображений» [24] архитектура ResNet стала основополагающей работой, продемонстрировавшей, что чрезвычайно глубокие сети можно обучать с использованием стандартного SGD и разумной функции инициализации. Для обучения сетей на глубине более 50–100 (а в некоторых случаях и 1000) слоев ResNet использует микроархитектуру, называемую остаточным модулем.

Еще одним интересным компонентом ResNet является то, что слои пула используются крайне экономно. Основываясь на работе Springenberg et al. [41], ResNet не полагается строго на максимальное количество операций объединения для уменьшения размера тома. Вместо этого свертки с шагом > 1 используются не только для обучения

веса, но уменьшить пространственные размеры выходного объема. Фактически, в полной реализации архитектуры применяется только два случая объединения:

1. Первая (и единственная) валюта максимального пулинга возникает на ранней стадии сети, чтобы помочь уменьшить пространственные размеры.
2. Вторая операция объединения на самом деле является средним слоем объединения, используемым вместо полного связанные слои, как в GoogLeNet.

Строго говоря, существует только один максимальный объединяющий слой — все остальные сокращения пространственных измерений обрабатываются сверточными слоями.

В этом разделе мы рассмотрим исходный остаточный модуль, а также остаточный модуль узкого места, используемый для обучения более глубоких сетей. Оттуда мы обсудим расширения и обновления исходного остаточного модуля He et al. в их публикации 2016 года «Отображение идентичности в глубоких остаточных сетях» [33], что позволяет нам еще больше повысить точность классификации. Позже в этой главе мы реализуем ResNet с нуля, используя Keras.

12.1.1 Углубление: остаточные модули и узкие места

Исходный остаточный модуль, предложенный He et al. в 2015 году, полагается на сопоставления идентификаторов, процесс передачи исходных данных в модуль и добавления их к результатам серии операций. Графическое изображение этого модуля можно увидеть на рис. 12.1 (слева). Обратите внимание, что этот модуль имеет только две ветви, в отличие от четырех ветвей в модуле Inception в GoogLeNet. Кроме того, этот модуль очень упрощен.

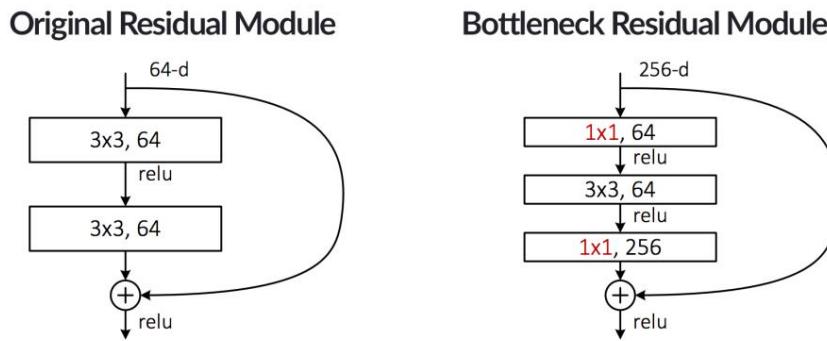


Рисунок 12.1: Слева: исходный остаточный модуль, предложенный He et al. Справа: наиболее часто используемый узкий вариант модуля Residual.

В верхней части модуля мы принимаем ввод в модуль (т. е. предыдущий уровень в сети). Правая ветвь является «линейным ярлыком» — она соединяет вход с операцией сложения в нижней части модуля. Затем к левой ветви остаточного модуля мы применяем серию сверток (все они 3×3), активации и нормализации пакетов. Это довольно стандартный шаблон, которому нужно следовать при построении сверточных нейронных сетей.

Но что делает ResNet интересным, так это то, что He et al. предложил добавить исходный ввод к выводу слоев CONV, RELU и BN. Мы называем это добавление тождественным отображением, поскольку вход (тождество) добавляется к выходу серии операций. Именно поэтому используется термин «остаточное». «Остаточные» входные данные добавляются к выходным данным серии операций со слоями. Соединение между входным и дополнительным узлом называется ярлыком. Обратите внимание, что мы не имеем в виду конкатенацию по измерению канала, как мы это делали в предыдущих главах. Вместо этого мы выполняем простое сложение $1 + 1 = 2$ в нижней части модуля между двумя ветвями.

В то время как традиционные слои нейронной сети можно рассматривать как изучающие функцию $y = f(x)$, остаточный слой пытается аппроксимировать y через $f(x) + id(x) = f(x) + x$, где $id(x)$ — это функция тождества.

Эти остаточные слои начинаются с функции идентификации и развиваются, становясь более сложными по мере обучения сети. Этот тип остаточной структуры обучения позволяет нам обучать сети, которые существенно глубже, чем ранее предложенные сетевые архитектуры.

Кроме того, поскольку ввод включен в каждый остаточный модуль, оказывается, что сеть может обучаться быстрее и с большей скоростью обучения. Очень часто можно увидеть, что базовая скорость обучения для реализаций ResNet начинается с $1e-1$. Для большинства архитектур, таких как AlexNet или VGGNet, такая высокая скорость обучения почти гарантирует, что сеть не сойдетсся. Но поскольку ResNet полагается на остаточные модули через сопоставления идентификаторов, такая более высокая скорость обучения вполне возможна.

В той же работе 2015 г. He et al. также включал расширение исходного остаточного модуля, называемое узкими местами (рис. 12.1, справа). Здесь мы видим, что происходит то же самое сопоставление идентичности, только теперь слои CONV в левой ветви остаточного модуля были обновлены:

1. Мы используем три слоя CONV, а не только два.

2. Первый и последний слои CONV представляют собой свертки 1×1 .

3. Количество фильтров, изученных на первых двух слоях CONV, составляет $1/4$ количества изученных фильтров в финальном конв.

Чтобы понять, почему мы называем это «узким местом», рассмотрим следующий рисунок, где два остатка модули укладываются друг на друга, при этом один остаток переходит в другой (рис. 12.2).

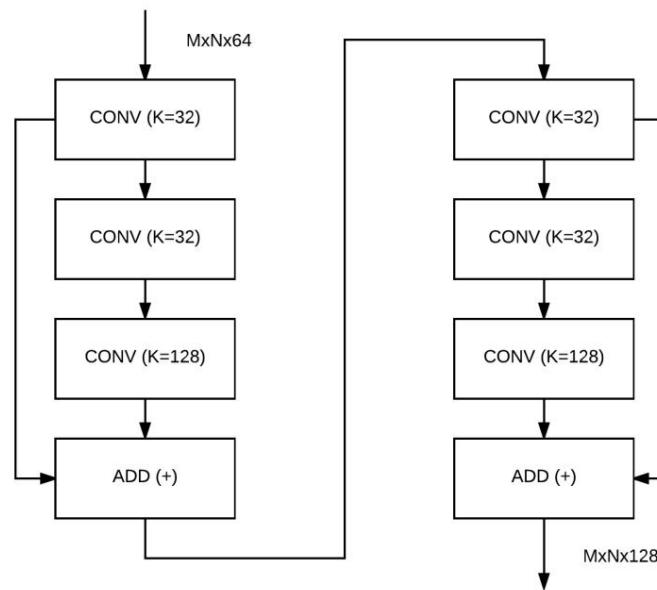


Рисунок 12.2: Пример двух сложенных остаточных модулей, где один переходит в другой. Оба модуля изучают $K = 32, 32$ и 128 фильтров соответственно. Обратите внимание, как размерность уменьшается на первых двух слоях CONV, а затем увеличивается на последнем слое CONV.

Первый остаточный модуль принимает входной объем размером $M \times N \times 64$ (фактическая ширина и высота для этого примера произвольны). Три слоя CONV в первом остаточном модуле изучают $K = 32, 32$ и 128 фильтров соответственно. После применения первого остаточного модуля размер нашего выходного объема составляет $M \times N \times 128$, который затем подается во второй остаточный модуль.

Во втором остаточном модуле количество фильтров, изученных каждым из трех слоев CONV, остается неизменным при $K = 32, 32$ и 128 соответственно. Однако обратите внимание, что $32 < 128$, что означает, что мы фактически уменьшаем размер тома в слоях 1×1 и 3×3 CONV. Преимущество этого результата заключается в том, что слой узкого места 3×3 остается с меньшими входными и выходными размерами.

Последнее преобразование 1×1 затем применяется в 4 раза больше фильтров, чем первые два слоя преобразования, таким образом

еще раз увеличивая размерность, поэтому мы называем это обновление остаточного модуля методом «бутилочного горлышка». При создании наших собственных остаточных модулей обычно используется псевдокод, такой как остаточный_модуль ($K = 128$) , который подразумевает, что последний слой CONV будет изучать 128 фильтров, а первые два будут изучать $128/4 = 32$ фильтра. С этой нотацией часто легче работать , поскольку понятно, что узкие места в слоях CONV изучают 1/4 количества фильтров в качестве конечного слоя CONV.

Когда дело доходит до обучения ResNet, мы обычно используем узкий вариант остаточного модуля, а не исходную версию, особенно для реализаций ResNet с > 50 уровнями.

12.1.2 Переосмысление остаточного модуля

В 2016 году He и соавт. опубликовал вторую статью об остаточном модуле под названием «Отображение идентичности в глубоких остаточных сетях» [33]. В этой публикации описано всестороннее исследование, как теоретическое, так и эмпирическое, упорядочения слоев свертки, активации и пакетной нормализации внутри самого остаточного модуля. Первоначально остаточный модуль (с узким местом) выглядел так, как показано на рис. 12.3 (слева).

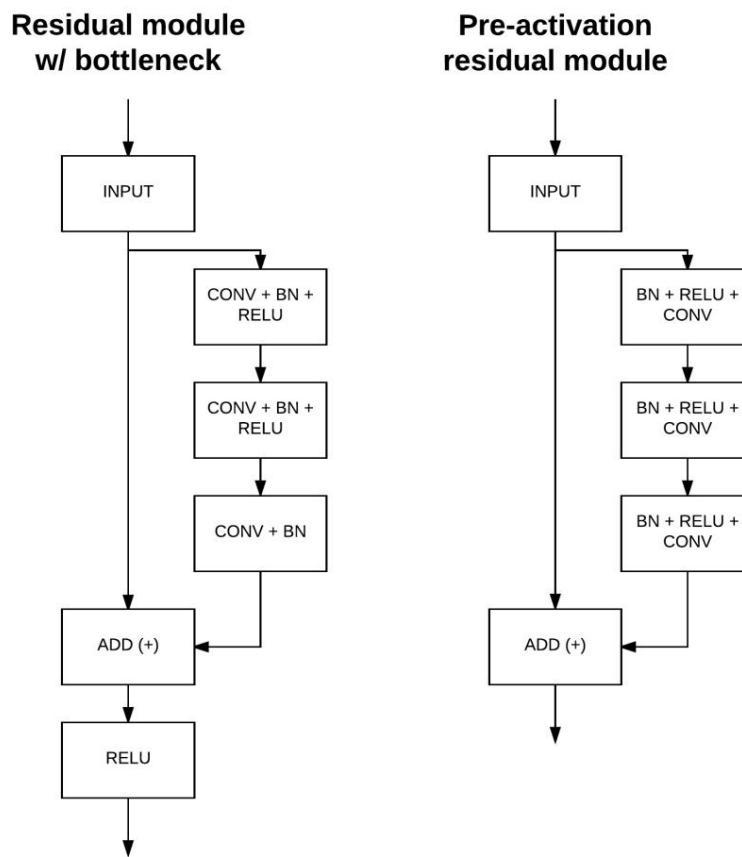


Рисунок 12.3: Слева: Исходный остаточный модуль с узким местом. Справа: адаптация модуля узкого места для использования предварительных активаций.

Исходный остаточный модуль с узким местом принимает ввод (карта активации ReLU), а затем применяет серию ($\text{CONV} \Rightarrow \text{BN} \Rightarrow \text{RELU} \times 2 \Rightarrow \text{CONV} \Rightarrow \text{BN}$ перед добавлением этого вывода к исходному вводу и применением окончательного Активации ReLU (которая затем передается в следующий остаточный модуль в сети). Однако He et al. В исследовании 2016 года было обнаружено более оптимальное упорядочение слоев, позволяющее получить более высокую точность — этот метод называется предварительной активацией.

В версии до активации остаточного модуля мы удаляем ReLU в нижней части модуль и переупорядочить нормализацию и активацию пакета так, чтобы они выполнялись до свертка (рис. 12.3, справа).

Теперь вместо того, чтобы начинать со свертки, применяем ряд (БН => РЕЛУ => КОНВ)* 3 (конечно, при условии, что узкое место используется). Выход остаточного модуля теперь операция сложения, которая затем передается следующему остаточному модулю в сети (поскольку оставшиеся модули укладываются друг на друга).

Мы называем этот уровень упорядочения предварительной активацией, поскольку наши ReLU и нормализация партии размещаются до сверток, что отличается от типичного подхода применения ReLU и пакетного нормализации после сверток. В следующем разделе мы реализуем ResNet с нуля.

использование как узких мест, так и предварительных активаций.

12.2 Внедрение ResNet

Теперь, когда мы рассмотрели архитектуру ResNet, давайте приступим к реализации в Keras. За этой конкретной реализацией, мы будем использовать самое последнее воплощение остаточного модуля, включая узкие места и предварительные активации. Чтобы обновить структуру проекта, создайте новый файл с именем resnet.py внутри подмодуля nn.conv pyimagesearch — именно здесь наш ResNet реализация будет жить:

```
-- pyimagesearch
| |--- __init__.py
| |--- обратные вызовы
| |--- ио
| |--- нн
| | |--- __init__.py
| | |--- конв.
| | | |--- __init__.py
| | | |--- alexnet.py
| | | --- более глубокий googlenet.py
| | | |--- lenet.py
| | | |--- minigooglenet.py
| | | |--- minivggnet.py
| | | |--- fheadnet.py
| | | |--- resnet.py
| | | |--- мелкая сеть.py
| |--- предварительная обработка
| |--- утилиты
```

Оттуда откройте resnet.py и вставьте следующий код:

```
1 # импортируем необходимые пакеты
2 из keras.layers.normalization import BatchNormalization
3 из keras.layers.convolutional импорт Conv2D
4 из keras.layers.convolutional import AveragePooling2D
5 из keras.layers.convolutional импортировать MaxPooling2D
6 из keras.layers.convolutional импортировать ZeroPadding2D
7 из keras.layers.core импорт Активация
8 из keras.layers.core импорт Плотный
9 из keras.layers import Flatten
10 из keras.layers import Input
11 из импорта keras.models Модель
```

```
12 из keras.layers импортировать добавить
13 из keras.regularizers импорт l2
14 из keras импортировать бэкэнд как K
```

Мы начинаем с импорта нашего довольно стандартного набора классов и функций при создании Сверточные нейронные сети. Однако я хотел бы обратить ваше внимание на строку 12 , где мы импортируем функцию добавления . Внутри остаточного модуля нам нужно сложить выходные данные двух ветвей, которые будут выполняться с помощью этого метода добавления . Мы также импортируем функцию l2 на Стока 13 , чтобы мы могли выполнить уменьшение веса L2. Регуляризация чрезвычайно важна, когда обучение ResNet, поскольку из-за глубины сети она склонна к переоснащению.

Далее, давайте перейдем к нашему остаточному_модулю:

```
16 класс Resnet:
17     @статический метод
18     определение остаточного_модуля (данные, K, шаг, chanDim, красный = False,
19         reg=0,0001, bnEps=2e-5, bnMom=0,9):
```

Эта конкретная реализация ResNet была вдохновлена He et al. в их дистрибутиве Caffe [46], а также в реализации mxnet от Wei Wu [47], поэтому мы будем следовать их выбор параметров как можно точнее. Глядя на остаточный_модуль , мы видим, что функция принимает больше параметров, чем любая из наших предыдущих функций — давайте рассмотрим каждую из них в деталь.

Параметр данных — это просто ввод в остаточный модуль. Значение K определяет количество фильтров, которые будут изучены окончательным преобразованием в узком месте. Первые два слоя CONV будут выучить фильтры K/4 , согласно He et al. бумага. Шаг управляет шагом свертки. Мы будем использовать этот параметр, чтобы помочь нам уменьшить пространственные размеры нашего объема, не прибегая к максимальный пул.

Затем у нас есть параметр chanDim , который определяет ось, которая будет выполнять пакетную нормализацию — это значение указывается позже в функции сборки в зависимости от того, используем ли мы «каналы» . последним» или «каналы первыми» .

Не все остаточные модули будут отвечать за уменьшение размеров нашего пространственного объема — красное (т.е. «уменьшить») логическое значение будет контролировать, уменьшаем ли мы пространственные измерения (Истина) или нет (ложь).

Затем мы можем предоставить силу регуляризации всем слоям CONV в остаточном модуле через рег. Параметр bnEps управляет ϵ , отвечающим за предотвращение ошибок «деления на ноль» при нормализации входных данных. В Keras значение ϵ по умолчанию равно 0,001; однако для нашей конкретной реализации мы будем позволять значительно уменьшить это значение. bnMom управляет импульсом движения в среднем. Это значение обычно по умолчанию равно 0,99 внутри Keras, но He et al. а также Вэй Ву рекомендуют уменьшить значение до 0,9.

Теперь, когда параметры остаточного_модуля определены, давайте перейдем к телу модуля. функция:

```
20     # ярлык модуля ResNet должен быть
21     # инициализировать как входные (идентификационные) данные
22     ярлык = данные
23
24     # первый блок модуля ResNet - это 1x1 CONV
25     bn1 = пакетная нормализация (ось = chanDim, epsilon = bnEps,
26         импульс = bnMom) (данные)
27     act1 = Активация ("relu") (bn1)
```

```
28         conv1 = Conv2D(int(K * 0,25), (1, 1), use_bias=False,
29                     kernel_regularizer=l2(reg))(act1)
```

В строке 22 мы инициализируем ярлык в остаточном модуле, который является просто ссылкой на входные данные. Позже мы добавим ярлык на выход нашей ветки узкого места + предактивации.

Первую предварительную активацию узкого места можно увидеть в строках 25-29. Здесь мы применяем слой пакетной нормализации, за которым следует активация ReLU, а затем свертка 1×1 с использованием всего фильтров $K/4$. Вы также заметите, что мы исключаем член смещения из наших слоев CONV через `use_bias = Ложь`. Почему мы можем намеренно опустить термин смещения? По словам Хе и др., смещения находятся в слоях BN, которые следуют сразу за свертками [48], поэтому нет необходимости ввести второй член смещения.

Затем у нас есть второй слой CONV в узком месте, он отвечает за изучение общего фильтров $K/4$, 3×3 :

```
31     # второй блок модуля ResNet - это 3x3 CONV
32     bn2 = Пакетная нормализация (ось = chanDim, epsilon = bnEps,
33             импульс=bnMom)(conv1)
34     act2 = Активация ("relu") (bn2)
35     conv2 = Conv2D(int(K * 0,25), (3, 3), шаги=шаг,
36             отступ = «то же», use_bias = ложь,
37             kernel_regularizer=l2(reg))(act2)
```

Последний блок в узком месте изучает K фильтров, каждый из которых имеет размер 1×1 :

```
39     # третий блок модуля ResNet - еще один набор 1x1
40     # КОНВ.
41     bn3 = Пакетная нормализация (ось = chanDim, epsilon = bnEps,
42             импульс=bnMom)(conv2)
43     act3 = Активация ("relu") (bn3)
44     conv3 = Conv2D(K, (1, 1), use_bias=False,
45                 kernel_regularizer=l2(reg))(act3)
```

Подробнее о том, почему мы называем это «узким местом» с «предактивацией», см. в разделе 12.1 выше.

Следующий шаг — посмотреть, нужно ли нам уменьшить пространственные измерения, тем самым облегчив необходимость применить максимальный пул:

```
47     # если мы хотим уменьшить пространственный размер, применяем слой CONV к
48     # ярлык
49     если красный:
50         ярлык = Conv2D(K, (1, 1), шаги=шаг,
51                         use_bias = False, kernel_regularizer = l2 (reg)) (act1)
```

Если нам поручат уменьшить пространственные измерения, мы сделаем это с помощью сверточного слоя (примененного к ярлыку) с шагом > 1 .

Выход окончательного `conv3` в узком месте добавляется вместе с ярлыком, таким образом выступающий в качестве вывода остаточного_модуля:

```

53         # складываем вместе ярлык и конечный CONV
54         x = добавить ([conv3, ярлык])
55
56         # вернуть дополнение в качестве вывода модуля ResNet
57         вернуть x

```

остаточный_модуль будет служить нашим строительным блоком при создании глубоких остаточных сетей .
Давайте перейдем к использованию этого строительного блока внутри метода сборки:

```

59     @статический метод
60     def build(ширина, высота, глубина, классы, этапы, фильтры,
61               reg=0.0001, bnEps=2e-5, bnMom=0.9, набор данных="cifar"):

```

Точно так же, как наш остаточный_модуль требует больше параметров, чем предыдущая микроархитектура реализации, то же верно и для нашей функции сборки . Классы ширины , высоты и глубины все они контролируют входные пространственные размеры изображений в нашем наборе данных. Переменная class диктует сколько общих классов должна выучить наша сеть — эти переменные вы уже видели.

Что интересно, так это параметры этапов и фильтров , оба из которых являются списками. Когда при построении архитектуры ResNet мы будем размещать ряд остаточных модулей поверх каждого другой (с использованием одинакового количества фильтров для каждого стека) с последующим уменьшением пространственных размеров объема — этот процесс затем продолжается до тех пор, пока мы не будем готовы применить наше среднее объединение и классификатор softmax.

Чтобы прояснить этот момент, давайте предположим, что этапы = (3, 4, 6) и фильтры = (64, 128, 256, 512). Первое значение фильтра, 64, будет применено к единственному слою CONV , не являющемуся частью остатка. модуль (т. е. первый сверточный слой в сети). Затем мы сложим три остаточных модуля на друг над другом — каждый из этих остаточных модулей выучит K = 128 фильтров. Пространственные размеры объема будет уменьшено, а затем мы перейдем ко второй записи поэтапно , где мы будем сложить четыре остаточных модуля друг на друга, каждый из которых отвечает за обучение K = 256 фильтров. После этих четырех остаточных модулей мы снова уменьшим размерность и перейдем к последней записи. в списке этапов , предписывая нам сложить шесть остаточных модулей друг над другом, где каждый остаточный модуль узнает K = 512.

Преимущество указания этапов и фильтров в списке (вместо их жесткого кодирования) заключается в том, что мы можем легко использовать циклы for для построения очень глубоких сетевых архитектур без введение раздувания кода — этот момент станет более ясным позже в нашей реализации.

Наконец, у нас есть параметр набора данных , который считается строкой. В зависимости от набор данных, для которого мы создаем ResNet, мы можем захотеть применить больше/меньше сверток и пакетных нормализаций, прежде чем мы начнем складывать наши остаточные модули. Мы увидим, почему мы можем захотеть изменить количество сверточных слоев в разделе 12.5 ниже, а пока можно смело игнорировать этот параметр.

Далее, давайте инициализируем наши inputShape и chanDim в зависимости от того, используем ли мы «каналы «последними» (строки 64 и 65) или «каналы первыми» (строки 69–71) .

```

62     # инициализируем входную форму как "последние каналы" и
63     # сам размер каналов
64     inputShape = (высота, ширина, глубина)
65     ЧанДим = -1
66
67     # если мы используем "сначала каналы", обновите форму ввода
68     # и размер каналов

```

```

69     если K.image_data_format() == "channels_first":
70         inputShape = (глубина, высота, ширина)
71         ЧанДим = 1

```

Теперь мы готовы определить ввод для нашей реализации ResNet:

```

73     # устанавливаем ввод и применяем BN
74     входы = ввод (форма = форма ввода)
75     x = пакетная нормализация (ось = chanDim, epsilon = bnEps,
76     импульс = bnMom) (входы)
77
78     # проверяем, используем ли мы набор данных CIFAR
79     если набор данных == "cifar":
80         # применить один слой CONV
81         x = Conv2D (фильтры [0], (3, 3), use_bias = False,
82                     padding="то же самое", kernel_regularizer=l2(reg))(x)

```

В отличие от предыдущих сетевых архитектур, которые мы видели в этой книге (где первый уровень обычно это CONV), мы видим, что ResNet использует BN в качестве первого уровня. Причина применения пакетная нормализация вашего ввода — это дополнительный уровень нормализации. На самом деле, выполнение партии нормализация на самом входе иногда может устраниć необходимость применения нормализации среднего значения к входы. В любом случае BN в строках 75 и 76 действует как дополнительный уровень нормализации.

Оттуда мы применяем один слой CONV к строкам 81 и 82. Этот слой CONV изучит общее количество фильтров[0], 3x 3 фильтра (имейте в виду, что фильтры — это список, поэтому это значение указывается через метод сборки при построении архитектуры).

Вы также заметите, что я проверил, использую ли мы набор данных CIFAR-10 (строка 79). Позже в этой главе мы обновим этот блок if , включив в него оператор elif для Tiny ImageNet. Поскольку входные размеры Tiny ImageNet больше, мы применим ряд сверток, нормализации и максимальное объединение (единственное максимальное объединение в архитектуре ResNet), прежде чем мы начнем укладка остаточных модулей. Однако пока мы используем только набор данных CIFAR-10.

Давайте продолжим и начнем укладывать оставшиеся слои друг на друга, краевый камень Resnet-архитектура:

```

84     # цикл по количеству стадий
85     для i в диапазоне (0, len (этапы)):
86         # инициализируем шаг, затем применяем остаточный модуль
87         # используется для уменьшения пространственного размера входного объема
88         шаг = (1, 1), если я == 0 иначе (2, 2)
89         x = ResNet.residual_module(x, фильтры[i + 1], шаг,
90                         chanDim, red=True, bnEps=bnEps, bnMom=bnMom)
91
92         # цикл по количеству слоев в сцене
93         для j в диапазоне (0, этапы [i] - 1):
94             # применить модуль ResNet
95             x = ResNet.residual_module(x, фильтры[i + 1],
96                         (1, 1), chanDim, bnEps=bnEps, bnMom=bnMom)

```

В строке 85 мы начинаем перебирать список этапов. Имейте в виду, что каждая запись в список этапов представляет собой целое число, указывающее, сколько остаточных модулей будет уложено поверх каждого другие. Следуя работе Спрингенберга и др., ResNet пытается сократить использование пулов, поскольку насколько это возможно, полагаясь на слои CONV для уменьшения пространственных размеров объема.

Чтобы уменьшить размер тома без объединения слоев, мы должны установить шаг свертки на Стока 88. Если это первая запись на этапе, мы установим шаг равным (1, 1), указывая, что не следует выполнять понижающую дискретизацию. Однако для каждого последующего этапа мы будем применять остаток модуль с шагом (2, 2), что позволит нам уменьшить размер тома.

Оттуда мы будем перебирать количество слоев на текущем этапе в строке 93 (т. е. количество остаточных модулей, которые будут уложены друг на друга). Количество фильтров в каждом остаточном модуле узнает, контролируется соответствующей записью в списке фильтров . Причина мы используем $i + 1$ в качестве индекса в фильтрах , потому что первое значение фильтра использовалось в строках 81 и 82. Остальные значения фильтра соответствуют количеству фильтров на каждом этапе. Как только у нас есть сложенные остаточные модули stage [i] друг над другом, наш цикл for возвращает нас к Lines 88-90 , где мы уменьшаем пространственные размеры объема и повторяем процесс.

На данный момент размер нашего тома был уменьшен до $8 \times 8 \times$ классов (вы можете проверить это для самостоятельно, вычислив размеры входных/выходных объемов для каждого слоя, или еще лучше, просто используя `plot_model` из главы 19 Starter Bundle).

Чтобы избежать использования плотных полно связанных слоев, вместо этого мы применим средний пул к уменьшите размер тома до $1 \times 1 \times$ классов:

```

98      # применяем BN => ACT => POOL
99      x = пакетная нормализация (ось = chanDim, epsilon = bnEps,
100         импульс=bnMom)(x)
101      x = активация («релю») (x)
102      x = средний пул2D ((8, 8)) (x)

```

Оттуда мы создаем плотный слой для общего количества классов , которые мы собираемся изучить, с последующим применением активации softmax для получения наших окончательных выходных вероятностей:

```

104      # классификатор softmax
105      x = сгладить () (x)
106      x = плотный (классы, kernel_regularizer = l2 (reg)) (x)
107      x = Активация ("softmax") (x)
108
109      # создаем модель
110      модель = модель (входы, x, имя = "resnet")
111
112      # вернуть построенную сетевую архитектуру
113      модель возврата

```

Затем полностью построенная модель ResNet возвращается вызывающей функции в строке 113.

12.3 ResNet на CIFAR-10

Помимо обучения меньших вариантов в ResNet на полном наборе данных ImageNet, я никогда не пытался для обучения ResNet работе с CIFAR-10 (или Стенфордскому испытанию Tiny ImageNet, как мы увидим в этом разделе). Из-за этого факта я решил рассматривать этот и следующий разделы как откровенные тематические исследования, в которых я раскрыть мои личные эмпирические правила и лучшие практики, которые я разработал за годы обучения нейронных сетей. сети.

Эти лучшие практики позволяют мне подходить к новой проблеме с первоначальным планом, повторять ее и в конечном итоге прийти к решению, которое получает хорошую точность. В случае с CIFAR-10 мы сможем повторить работу He et al. и претендовать на место среди других современных подходов [49].

12.3.1 Обучение ResNet на CIFAR-10 методом ctrl+c

Всякий раз, когда я начинаю новый набор экспериментов либо с незнакомой мне сетевой архитектурой, либо с набором данных, с которым я никогда не работал, либо с тем и другим, я всегда начинаю с метода обучения `ctrl + c`. Используя этот метод, я могу начать обучение с начальной скоростью обучения (и соответствующим набором гиперпараметров), отслеживать обучение и быстро корректировать скорость обучения на основе результатов по мере их поступления. Этот метод особенно полезен, когда я совершенно не уверен в приблизительное количество эпох, которое потребуется данной архитектуре для получения разумной точности или определенного набора данных.

В случае с CIFAR-10 у меня есть предыдущий опыт (как и у вас, после прочтения всех остальных глав этой книги), поэтому я вполне уверен, что это займет 60-100 эпох, но точно не уверен так как я никогда раньше не тренировал ResNet на CIFAR-10.

Поэтому наши первые несколько экспериментов будут основываться на методе обучения `ctrl + c`, чтобы сузить круг гиперпараметров, которые мы должны использовать. Как только мы освоимся с нашим набором гиперпараметров, мы переключимся на определенный график снижения скорости обучения в надежде выжить из процесса обучения все до последнего кусочка точности.

Для начала откройте новый файл, назовите его `resnet_cifar10.py` и вставьте следующий код:

```
1 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом режиме
2 import matplotlib 3 matplotlib.use("Agg")
```

4

```
5 # импортируем необходимые пакеты 6
из sklearn.preprocessing импортируем LabelBinarizer 7 из
pyimagesearch.nn.conv импортируем ResNet 8 из
pyimagesearch.callbacks импортируем EpochCheckpoint 9 из
pyimagesearch.callbacks импортируем TrainingMonitor 10 из
keras.preprocessing.image импортируем ImageDataGenerator 11 из
keras.optimizers импортируем SGD 12 из keras.datasets импортируем
cifar10 13 из keras.models импортируем load_model 14 импортируем
keras.backend как K 15 импортируем numpy как np 16 импортируем
argparse 17 импортировать sys
```

18

```
19 # установить высокий предел рекурсии, чтобы Theano не жаловался
20 sys.setrecursionlimit(5000)
```

Мы начинаем с импорта необходимых пакетов Python в строках 6-17. Поскольку мы будем использовать метод `ctrl + c` для обучения, мы обязательно импортируем класс `EpochCheckpoint` (строка 8) для сериализации весов ResNet на диск во время процесса обучения, что позволит нам останавливать и возобновлять обучение с определенной контрольной точки. . Поскольку это наш первый эксперимент с ResNet, мы будем использовать оптимизатор `SGD` (строка 11) — время покажет, решим ли мы переключиться и использовать другой оптимизатор (мы позволим нашим результатам определять это).

В строке 20 я обновляю лимит рекурсии для языка программирования Python. Я написал эту книгу, имея в виду как TensorFlow, так и Theano, поэтому, если вы используете TensorFlow, вам не нужно беспокоиться об этой строке. Однако, если вы используете Theano, вы можете столкнуться с ошибкой при создании экземпляра архитектуры ResNet, когда достигнут максимальный уровень рекурсии. Это известная «ошибка» в Theano, и ее можно устранить, просто увеличив предел рекурсии языка программирования Python [50].

Далее, давайте проанализируем наши аргументы командной строки:

```

22 # построить разбор аргумента и разобрать аргументы 23 ap =
argparse.ArgumentParser() 24 ap.add_argument("-c", "--checkpoints",
required=True,
25     help="путь к выходному каталогу контрольных точек")
26 ap.add_argument("-m", "--model", type=str, help="путь к
27     *конкретной* контрольной точке модели для загрузки") 28
ap.add_argument("-s", "--start-epoch", тип=целое, по умолчанию=0,
29     help="эпоха для возобновления обучения")
30 args = vars(ap.parse_args())

```

Нашему сценарию потребуется только переключатель `--checkpoints`, путь к каталогу, в котором мы будем хранить веса ResNet каждые N эпох. В случае, если нам нужно перезапустить обучение с определенной эпохи, мы можем указать путь `--model` вместе с целым числом, указывающим конкретный номер эпохи.

Следующим шагом является загрузка набора данных CIFAR-10 с диска (предварительно разделенного на обучение и тестирование), выполнение среднего вычитания и горячее кодирование целочисленных меток в виде векторов:

```

32 # загружаем обучающие и тестовые данные, конвертируя изображения из 33 #
целых чисел в числа с плавающей запятой 34 print("[INFO] загружаем данные
CIFAR-10...") 35 ((trainX, trainY), (testX, testY)) = cifar10.load_data() 36 trainX =
trainX.astype("плавающий") 37 testX = testX.astype("плавающий")

38
39 # применить к данным вычитание среднего 40
mean = np.mean(trainX, axis=0) 41 trainX -= mean

42 testX -= среднее
43
44 # преобразовать метки из целых чисел в векторы 45 lb =
LabelBinarizer() 46 trainY = lb.fit_transform(trainY) 47 testY =
lb.transform(testY)

```

Пока мы этим занимаемся, давайте также инициализируем `ImageDataGenerator`, чтобы мы могли применять увеличение данных. связь с CIFAR-10:

```

49 # построить генератор изображений для увеличения данных 50 aug
= ImageDataGenerator(width_shift_range=0.1,
51     height_shift_range=0.1, horizontal_flip=Истина,
52     fill_mode="ближайший")

```

В случае, если мы обучаем ResNet с самой первой эпохи, нам нужно создать экземпляр сети. архитектура:

```

54 # если не указана конкретная контрольная точка модели, то инициализируйте 55 #
сеть (ResNet-56) и скомпилируйте модель 56 если args["model"] равно None:
57     print("[INFO] компиляция модели...") opt =
58     SGD(lr=1e-1) model = ResNet.build(32, 32, 3,
59     10, (9, 9, 9),

```


Наконец, мы будем обучать нашу сеть партиями по 128 штук:

```

84 # обучаем сеть
85 print("[INFO] обучающая сеть...")
86 модель.fit_generator(
87     aug.flow(trainX, trainY, batch_size = 128),
88     validation_data=(тестX, тестY),
89     steps_per_epoch=len(trainX) // 128, эпох=10,
90     обратные вызовы = обратные вызовы, подробный = 1)

```

Теперь, когда наш скрипт `resnet_cifar10.py` написан, давайте перейдем к запуску экспериментов. с этим.

ResNet на CIFAR-10: Эксперимент №1

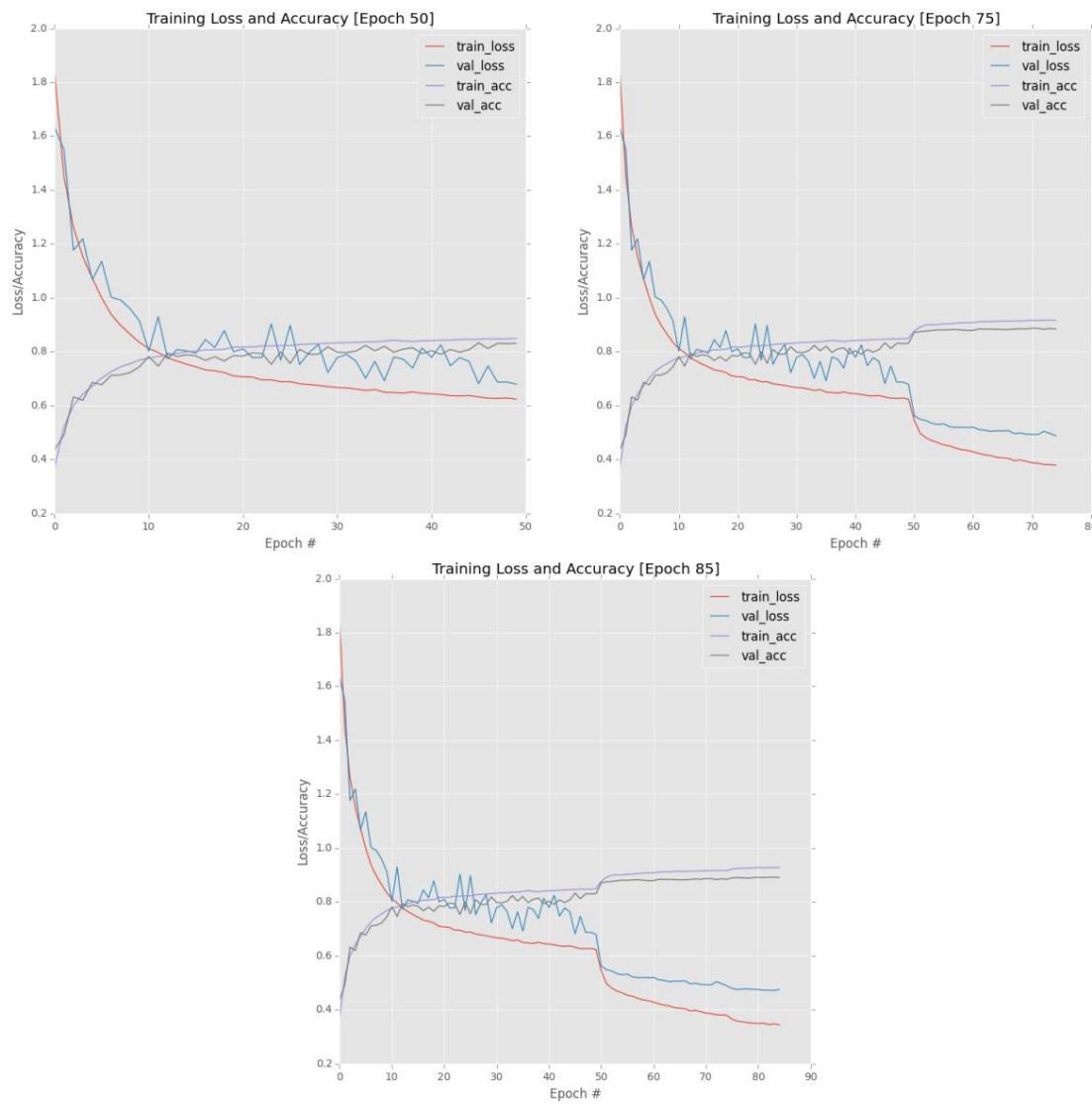


Рисунок 12.4: Вверху слева: первые 50 эпох при обучении ResNet на CIFAR-10 в эксперименте №1. Вверху справа: следующие 25 эпох. Внизу: последние 10 эпох.

В моем самом первом эксперименте с CIFAR-10 меня беспокоило количество фильтров в сети, особенно в отношении переобучения. По этой причине мой первоначальный список фильтров состоял из (16, 16, 32, 64) и (9, 9, 9) этапов остаточных модулей. Я также применил очень небольшое количество регуляризации L2 с $\text{reg} = 0,0001$ — я знал, что регуляризация потребуется, но я не был уверен в правильном количестве (пока). ResNet обучался с использованием SGD с базовой скоростью обучения $1e - 1$ и импульсом 0,9.

Я начал обучение, используя следующую команду:

```
$ python resnet_cifar10.py --вывод контрольных точек/контрольные точки
```

В прошлую эпоху 50 я заметил, что потери при обучении начинают замедляться, а также некоторая волатильность потерь при проверке (и растущий разрыв между ними) (рис. 12.4, вверху слева). Я прекратил обучение, снизил скорость обучения до $1e - 2$, а затем продолжил обучение:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints \ --model output/ checkpoints/epoch_50.hdf5 --start-epoch 50
```

Снижение скорости обучения оказалось очень эффективным, стабилизировав потери при проверке, но также начало появляться переобучение на обучающем наборе (что неизбежно при работе с CIFAR-10) примерно в эпоху 75 (рис. 12.4, вверху справа). После 75-й эпохи я снова прекратил обучение, снизил скорость обучения до $1e - 3$ и разрешил ResNet продолжать обучение еще 10 эпох:

```
$ python resnet_cifar10.py --checkpoints output/checkpoints \ --model output/ checkpoints/epoch_75.hdf5 --start-epoch 75
```

Окончательный график показан на рис. 12.4 (внизу), где мы достигаем точности 89,06 % на проверочном наборе. Для нашего самого первого эксперимента 89,06% — хорошее начало; однако это не так высоко, как 90,81%, достигнутые GoogLeNet в главе 11. Кроме того, He et al. сообщил о точности 93% с ResNet на CIFAR-10, поэтому нам явно есть над чем поработать.

12.3.2 ResNet на CIFAR-10: Эксперимент №2

В нашем предыдущем эксперименте была достигнута разумная точность 89,06%, но нам нужна более высокая точность. Вместо того, чтобы увеличивать глубину сети (путем добавления дополнительных этапов), я решил добавить дополнительные фильтры к каждому из слоев CONV. Таким образом, мой список фильтров был обновлен до (16, 64, 128, 256).

Обратите внимание, что количество фильтров во всех остаточных модулях удвоилось по сравнению с предыдущим экспериментом (количество фильтров в первом слое CONV осталось прежним). SGD снова использовался для обучения сети с импульсом 0,9. Я также сохранил срок регуляризации на уровне 0,0001.

На рисунке 12.5 (вверху слева) вы можете найти график моих первых 40 эпох: мы ясно видим разрыв между потерями при обучении и потерями при проверке, но в целом точность проверки по-прежнему соответствует точности обучения. Стремясь повысить точность, я решил снизить скорость обучения с $1e-1$ до $1e-2$ и тренироваться еще пять эпох — в результате обучение полностью застопорилось (вверху справа). Снижение скорости обучения снова с $1e-2$ до $1e-3$ даже вызвало переобучение из-за небольшого увеличения потерь при проверке (внизу)

Интересно, что потери не изменились как для обучающего набора, так и для проверочного набора, мало чем отличаясь от предыдущих экспериментов, которые мы проводили с GoogLeNet. Похоже, что после первоначального падения скорости обучения наша сеть больше не могла изучать какие-либо базовые шаблоны в наборе данных. Все это говорит о том, что после проверки 50-й эпохи точность увеличилась до 90,10%, что является улучшением по сравнению с нашим первым экспериментом.

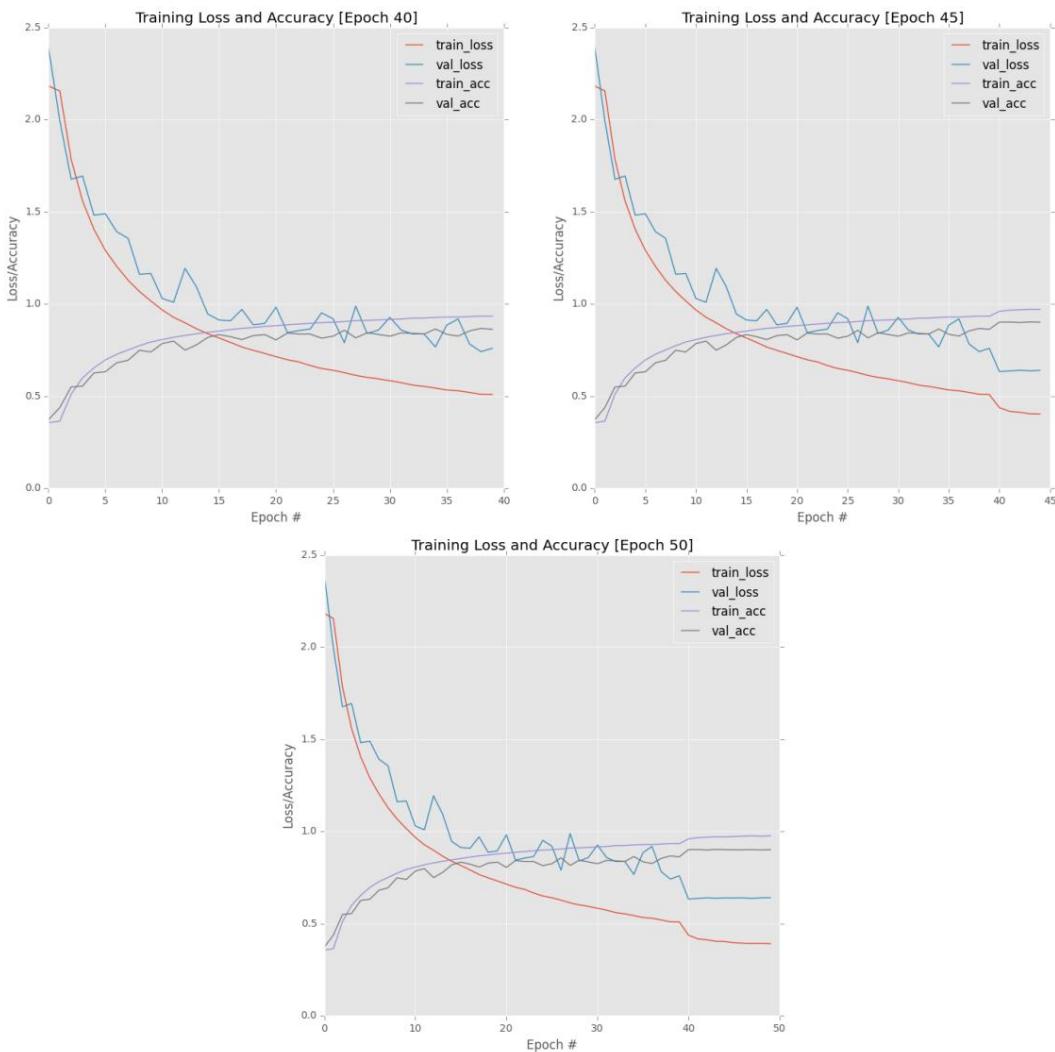


Рисунок 12.5: Вверху слева: первые 40 эпох при обучении ResNet на CIFAR-10 в эксперименте № 2. Вверху справа: следующие 5 эпох. Внизу: последние 5 эпох.

ResNet на CIFAR-10: эксперимент №3 В

этот момент мне стало более комфортно обучать ResNet на CIFAR-10. Очевидно , что увеличение количества фильтров помогло, но застой в обучении после первого падения скорости обучения все еще беспокоил. Я был уверен, что медленное линейное снижение скорости обучения поможет справиться с этой проблемой, но я не был уверен, что получил хороший набор гиперпараметров, оправдывающий переход на снижение скорости обучения.

Вместо этого я решил увеличить количество фильтров, изученных на первом слое CONV , до 64 (с 16), превратив список фильтров в (64, 64, 128, 256). Увеличение количества фильтров помогло во втором эксперименте, и нет причин, по которым первый слой CONV также должен упускать эти преимущества . Оптимизатор SGD остался один с начальной скоростью обучения $1e-1$ и импульсом 0,9.

Кроме того, я также решил резко увеличить регуляризацию с 0,0001 до 0,0005. У меня было подозрение, что если позволить сети обучаться дольше, это приведет к более высокой точности проверки — использование большего члена регуляризации, вероятно, позволит мне обучаться дольше.

Я также рассматривал возможность снижения скорости обучения, но, учитывая, что сеть делала

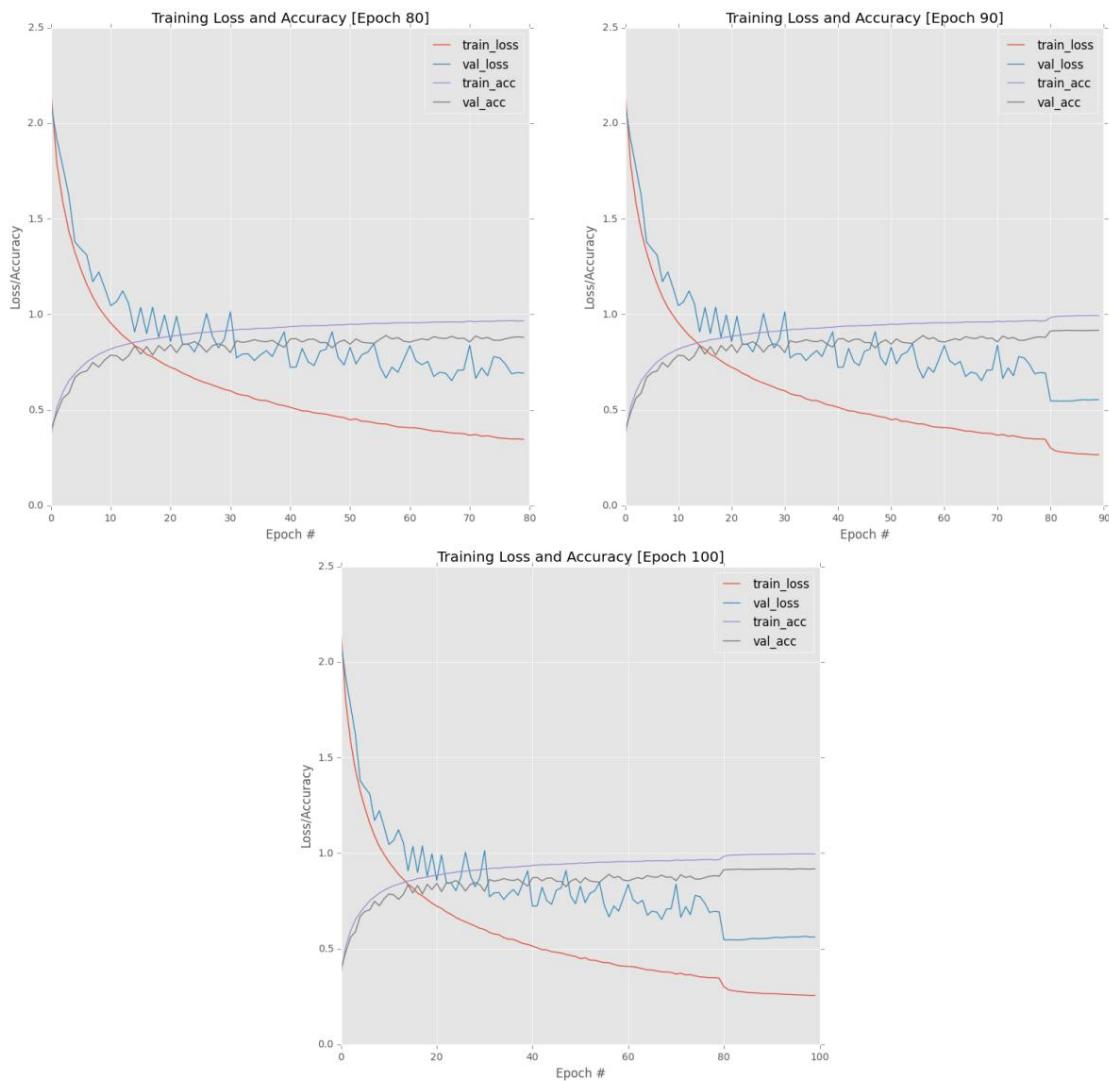


Рисунок 12.6: Вверху слева: первые 80 эпох при обучении ResNet на CIFAR-10 в эксперименте №3.

Вверху справа: следующие 10 эпох. Внизу: последние 10 эпох.

тяга без проблем на $1e-1$, вряд ли стоило снижать до $1e-2$. Это могло бы стабилизировать обучение (т. е. уменьшить колебания потерь/точности при проверке), но в конечном итоге привело бы к снижению точности после завершения обучения. Если бы мое подозрение о большей скорости обучения + большем сроке регуляризации оказалось верным, то имело бы смысл переключиться на затухание скорости обучения, чтобы избежать стагнации после падения порядка величины. Но прежде чем я смог сделать этот переход, мне сначала нужно было доказать свою догадку.

Как показывает мой график первых 80 эпох (рис. 12.6, вверху слева), переобучение определенно имеет место, поскольку проверка быстро расходится с обучением. Однако здесь интересно то, что, хотя переобучение, несомненно, имеет место (поскольку потери при обучении падают намного быстрее, чем потери при проверке), мы можем обучать сеть дольше без увеличения потерь при проверке.

После 80-й эпохи я прекратил обучение, снизил скорость обучения до $1e-2$, а затем тренировался еще 10 эпох (рис. 12.6, вверху справа). Мы видим начальное падение и потерю, а затем увеличение точности, но оттуда потери/точность проверки выходят на плато. Кроме того, мы можем начать видеть увеличение потерь при проверке, что является верным признаком переобучения.

Чтобы убедиться, что переобучение действительно имеет место, я прекратил обучение на 90-й эпохе, снизил скорость обучения до 1e-3, а затем тренировался еще 10 эпох (рис. 12.6, внизу). Конечно же, это явный признак переобучения: потери при проверке увеличиваются, а потери при обучении уменьшаются/остаются постоянными.

Однако, что очень интересно, так это то, что после 100-й эпохи мы получили точность проверки 91,83%, что выше, чем в нашем третьем эксперименте. Недостатком является то, что мы переоснащены — нам нужен способ поддерживать этот уровень точности (и повышать его) без переобучения. Для этого я решил перейти от обучения `ctrl + c` к уменьшению скорости обучения.

12.4 Обучение ResNet на CIFAR-10 с уменьшением скорости обучения

На данный момент кажется, что мы продвинулись настолько далеко, насколько это возможно, используя стандартное обучение `ctrl + c`. Мы также смогли увидеть, что наши самые успешные эксперименты происходят, когда мы можем тренироваться дольше, в диапазоне 80-100 эпох. Однако есть две основные проблемы, которые нам необходимо решить:

1. Всякий раз, когда мы снижаем скорость обучения на порядок и возобновляем обучение, мы получаем хороший скачок в точности, но затем мы быстро платим.

2. Мы переоснащаемся.

Чтобы решить эти проблемы и еще больше повысить точность, можно попробовать линейно уменьшить скорость обучения в течение большого количества эпох, как правило, примерно столько же, сколько ваши самые длинные эксперименты `ctrl + c` (если не немного дольше). Для начала давайте откроем новый файл, назовем его `resnet_cifar10_decay.py` и вставим следующий код:

```
1 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом режиме
2 import matplotlib 3 matplotlib.use("Agg")
```

4

```
5 # импортируем необходимые пакеты 6
из sklearn.preprocessing импортируем LabelBinarizer 7 из
pyimagesearch.nn.conv импортируем ResNet 8 из
pyimagesearch.callbacks импортируем TrainingMonitor 9 из
keras.preprocessing.image импортируем ImageDataGenerator 10 из
keras.callbacks импортируем LearningRateScheduler 11 из keras.optimizers
импорт SGD 12 из keras.datasets импорт cifar10 13 импорт numpy as np 14
импорт argparse 15 импорт sys 16 импорт os
```

17

```
18 # установить высокий предел рекурсии, чтобы Theano не жаловался 19
sys.setrecursionlimit(5000)
```

Строка 2 настраивает `matplotlib`, чтобы мы могли сохранять графики в фоновом режиме. Затем мы импортируем остальные наши пакеты Python в строках 6-16. Взгляните на строку 10, где мы импортируем наш `LearningRateScheduler`, чтобы мы могли определить пользовательское снижение скорости обучения для процесса обучения. Затем мы устанавливаем высокий предел системной рекурсии, чтобы избежать каких-либо проблем с серверной частью `Theano` (на всякий случай, если вы ее используете).

Следующим шагом является определение графика снижения скорости обучения:

```
21 # определить общее количество эпох для обучения вместе с 22 # начальной
скоростью обучения
```

```

23 NUM_EPOCHS = 100
24 INIT_LR = 1e-1
25
26 по определению poly_decay (эпоха):
27     # инициализируем максимальное количество эпох, базовую скорость обучения,
28     # и мощность полинома
29     maxEpochs = NUM_EPOCHS
30     baseLR = INIT_LR
31     мощность = 1,0
32
33     # вычислить новую скорость обучения на основе полиномиального распада
34     альфа = baseLR * (1 - (эпоха / число с плавающей запятой (maxEpochs))) ** мощность
35
36     # вернуть новую скорость обучения
37     вернуть альфа

```

Мы будем обучать нашу сеть в общей сложности 100 эпох с базовой скоростью обучения $1e - 1$. Функция `poly_decay` будет уменьшать нашу скорость обучения $1e - 1$ линейно в течение 100 эпох. Это линейное затухание из-за того, что мы установили `power=1`. Для получения дополнительной информации о скорости обучения **расписания см. в главе 16 стартового пакета вместе с главой 11 пакета для практикующих.**

Затем нам нужно указать два аргумента командной строки:

```

39 # построить аргумент parse и разобрать аргументы
40 ap = argparse.ArgumentParser()
41 ap.add_argument("-m", "--model", required=True,
42                 help="путь к выходной модели")
43 ap.add_argument("-o", "--output", required=True,
44                 help="путь к выходному каталогу (журналы, графики и т. д.)")
45 аргументов = переменные (ap.parse_args())

```

Переключатель `-model` управляет путем к нашей финальной сериализованной модели после обучения, а `-output` — это базовый каталог, в котором мы будем хранить любые журналы, графики и т. д.

Теперь мы можем загрузить набор данных CIFAR-10 и нормализовать его:

```

47 # загрузить данные обучения и тестирования, конвертируя изображения из
48 # целые числа в числа с плавающей запятой
49 print("[INFO] загрузка данных CIFAR-10...")
50 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
51 trainX = trainX.astype("плавающий")
52 testX = testX.astype("с плавающей запятой")
53
54 # применить к данным вычитание среднего
55 среднее значение = np.mean (trainX, ось = 0)
56 поездX -= среднее
57 testX -= среднее

```

Закодируйте целочисленные метки как векторы:

```

59 # преобразовать метки из целых чисел в векторы
60 фунтов = LabelBinarizer()
61 trainY = фунт.fit_transform(trainY)
62 testY = lb.transform(testY)

```

А также инициализируем наш ImageDataGenerator для аргументации данных:

```
64 # построить генератор изображений для увеличения данных
65 августа = ImageDataGenerator (width_shift_range = 0,1,
66     height_shift_range=0,1, horizontal_flip=Истина,
67     fill_mode="ближайший")
```

Наш список обратных вызовов будет состоять как из TrainingMonitor , так и из LearningRateScheduler . с функцией poly_decay в качестве единственного аргумента — этот класс позволит нам затухать скорость обучения, когда мы тренируемся.

```
69 # построить набор коллбэков
70 figPath = os.path.sep.join([args["output"], "{}.png".format(
71     os.getpid())])
72 jsonPath = os.path.sep.join([args["output"], "{}.json".format(
73     os.getpid())])
74 обратных вызова = [TrainingMonitor (figPath, jsonPath = jsonPath),
75     LearningRateScheduler (poly_decay)]
```

Затем мы создадим экземпляр ResNet с лучшими параметрами, которые мы нашли в разделе 12.6 (три стеки из девяти остаточных модулей: 64 фильтра в первом слое CONV перед остаточными модулями и 64, 128 и 256 фильтров для каждого стека соответствующих остаточных модулей):

```
77 # инициализируем оптимизатор и модель (ResNet-56)
78 print("[INFO] компилируется модель...")
79 opt = SGD(lr=INIT_LR, импульс=0,9)
80 модель = ResNet.build(32, 32, 3, 10, (9, 9, 9),
81     (64, 64, 128, 256), reg=0,0005)
82 модель.compile(loss="categorical_crossentropy", оптимизатор=opt,
83     метрики=["точность"])
```

Затем мы обучим нашу сеть, используя затухание скорости обучения:

```
85 # обучаем сеть
86 print("[INFO] обучающая сеть...")
87 модель.fit_generator(
88     aug.flow (trainX, trainY, batch_size = 128),
89     validation_data=(тестX, тестY),
90     steps_per_epoch=len(trainX) // 128, эпох=10,
91     обратные вызовы = обратные вызовы, подробный = 1)
92
93 # сохранить сеть на диск
94 print("[INFO] сериализуемая сеть...")
95 модель.save(аргументы["модель"])
```

Большой вопрос заключается в том, окупится ли снижение скорости обучения? Чтобы узнать, перейдите к следующему разделу.

ResNet на CIFAR-10: Эксперимент № 4 Как видно

из кода в предыдущем разделе, мы собираемся использовать оптимизатор SGD с базовой скоростью обучения $1e^{-1}$ и импульсом 0,9. Мы будем обучать ResNet в общей сложности 100 эпох, линейно уменьшая линейную скорость с $1e^{-1}$ до нуля. Чтобы обучить ResNet на CIFAR-10 с уменьшением скорости обучения, я выполнил следующую команду:

```
$ python resnet_cifar10_decay.py --output output\ --model
      output/resnet_cifar10.hdf5
```

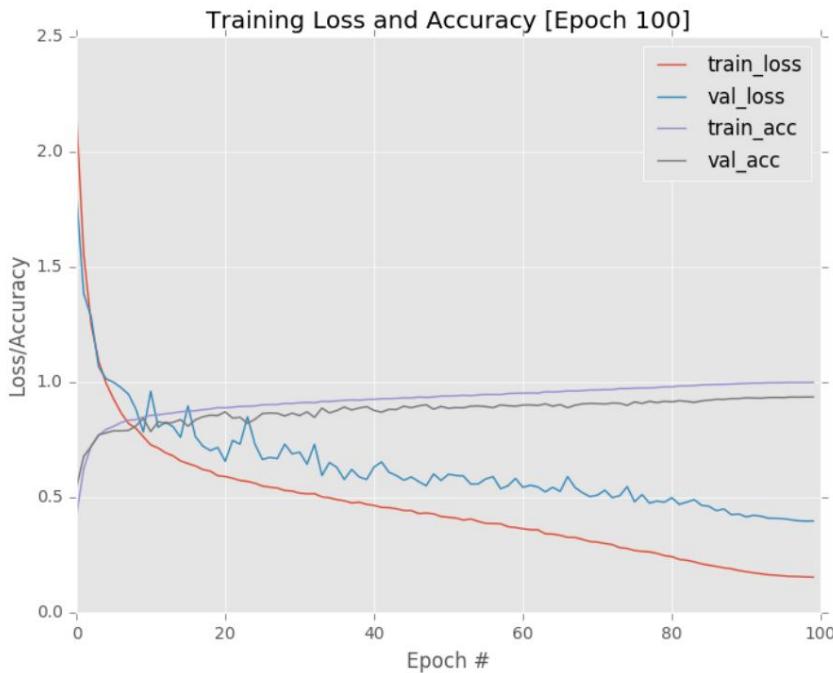


Рисунок 12.7: Обучение ResNet на CIFAR-10 с использованием снижения скорости обучения превосходит все предыдущие эксперименты.

После завершения обучения я взглянул на график (рис. 12.7). Как и в предыдущих экспериментах, потери при обучении и проверке начинают расходиться на ранней стадии, но, что более важно, разрыв остается примерно постоянным после первоначального расхождения. Этот результат важен, поскольку он указывает на то, что наше переоснащение находится под контролем. Мы должны смириться с тем, что при обучении на CIFAR-10 мы будем переобуваться, но нам нужно контролировать это переоснащение. Применив снижение скорости обучения, мы смогли успешно это сделать.

Вопрос в том, получили ли мы более высокую точность классификации? Чтобы ответить на это, взгляните на результат последних нескольких эпох:

...

Эпоха 98/100
247 с - потеря: 0,1563 - акк: 0,9985 - val_loss: 0,3987 - val_acc: 0,9351 Эпоха 99/100
245 с - потеря: 0,1548 - акк: 0,9987 - val_loss: 0,3973 - val_acc: 0,9100 Эпоха

244s - потеря: 0,1538 - акк: 0,9990 - val_loss: 0,3978 - val_acc: 0,9358 [INFO]
сериализация сети...

После 100-й эпохи ResNet достигает точности 93,58% на нашем тестовом наборе. Этот результат значительно выше, чем в наших предыдущих двух экспериментах, и, что более важно, он позволил нам воспроизвести результаты He et al. при обучении ResNet на CIFAR-10.

Взглянув на таблицу лидеров CIFAR-10 [49], мы видим, что He et al. достигла точности 93,57% [24], почти идентичной нашему результату (рис. 12.8). Красная стрелка указывает на нашу точность, благодаря чему мы благополучно попали в топ-10 лидеров.

Result	Method	Venue	Details
96.53%	Fractional Max-Pooling	arXiv 2015	Details
95.59%	Striving for Simplicity: The All Convolutional Net	ICLR 2015	Details
94.16%	All you need is a good init	ICLR 2016	Details
We obtained 93.58% accuracy, thus successfully replicating the work of He et al.	Lessons learned from manually classifying CIFAR-10	unpublished 2011	Details
94%	Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree	AISTATS 2016	Details
93.95%	Spatially-sparse convolutional neural networks	arXiv 2014	
93.72%	Scalable Bayesian Optimization Using Deep Neural Networks	ICML 2015	
93.63%	Deep Residual Learning for Image Recognition	arXiv 2015	Details
93.57%	Fast and Accurate Deep Network Learning by Exponential Linear Units	arXiv 2015	Details
93.45%	Universum Prescription: Regularization using Unlabeled Data	arXiv 2015	
93.34%	Batch-normalized Maxout Network in Network	arXiv 2015	
93.25%			

Рисунок 12.8: Мы успешно воспроизвели работу He et al. при применении ResNet к CIFAR-10 до 0,01%.

12.5 ResNet на Tiny ImageNet

В этом разделе мы обучим архитектуру ResNet (с узким местом и предварительной активацией) на вызове Stanford cs231n Tiny ImageNet. Подобно экспериментам ResNet + CIFAR-10 ранее в этой главе, я никогда раньше не обучал ResNet на Tiny ImageNet, поэтому я собираюсь применить тот же самый процесс эксперимента: 1. Начните с обучения на основе Ctrl + C, чтобы получить базовый уровень.

2. Если застой / плато происходит после падения скорости обучения на порядок, то переключитесь к снижению скорости обучения.

Учитывая, что мы уже применили подобную технику к CIFAR-10, мы должны сэкономить некоторое время, заметив признаки переобучения и выхода на плато раньше. Тем не менее, давайте начнем с рассмотрения структуры каталогов для этого проекта, которая почти идентична задаче GoogLeNet и Tiny ImageNet из предыдущей главы:

```
-- resnet_tinyimagenet.py | |--  
конфигурация | | |-- __init__.py
```

```

| |--- tiny_imagenet_config.py
|--- rank_accuracy.py
|--- train.py
|--- train_decay.py
|--- вывод/
| |--- КПП/
| |--- крошечное-изображение-net-200-mean.json

```

Здесь вы можете видеть, что мы создали модуль конфигурации Python, в котором мы сохранили файл с именем крошечный_имагенет_config.py.

Этот файл был скопирован непосредственно из главы GoogLeNet. Затем я обновил строки 31-39, чтобы указать к выходным данным ResNet MODEL_PATH, FIG_PATH (график обучения) и JSON_PATH (сериализованный журнал истории тренировок):

```

31 # определить путь к выходному каталогу, используемому для хранения графиков,
32 # классификационные отчеты и т. д.
33 OUTPUT_PATH = "выход"
34 MODEL_PATH = path.sep.join([OUTPUT_PATH,
35     "resnet_tinyimagenet.hdf5"])
36 FIG_PATH = path.sep.join([OUTPUT_PATH,
37     "resnet56_tinyimagenet.png"])
38 JSON_PATH = path.sep.join([OUTPUT_PATH,
39     "resnet56_tinyimagenet.json"])

```

Оттуда у нас есть train.py, который будет отвечать за обучение ResNet с использованием стандартного метода `ctrl+c`. В случае, если мы хотим применить уменьшение скорости обучения, мы сможем использовать train_decay.py. Наконец, rank_accuracy.py будет использоваться для вычисления ранга 1 и ранга 5. точность ResNet на Tiny ImageNet.

12.5.1 Обновление архитектуры ResNet

Ранее в этой главе мы подробно рассмотрели нашу реализацию архитектуры ResNet.

В частности, мы отметили параметр набора данных, предоставленный методу сборки, например:

```

59     @статический метод
60     def build(ширина, высота, глубина, классы, этапы, фильтры,
61               reg=0,0001, bnEps=2e-5, bnMom=0,9, набор данных="cifar"):

```

Это значение по умолчанию равно cifar; однако, поскольку мы сейчас работаем с Tiny ImageNet, мы необходимо обновить нашу реализацию ResNet, чтобы включить блок if/elif. Иди и открой ваш файл resnet.py в подмодуле nn.conv PyImageSearch и вставьте следующий код:

```

78     # проверяем, используем ли мы набор данных CIFAR
79     если набор данных == "cifar":
80         # применить один слой CONV
81         x = Conv2D (фильтры [0], (3, 3), use_bias = False,
82                     padding="то же самое", kernel_regularizer=l2(reg))(x)
83
84     # проверяем, используем ли мы набор данных Tiny ImageNet
85     набор данных elif == "tiny_imagenet":
86         # применяем CONV => BN => ACT => POOL для уменьшения пространственного размера

```

```

87         x = Conv2D(фильтры [0], (5, 5), use_bias = False,
88                     padding="same", kernel_regularizer=l2(reg))(x) x =
89         BatchNormalization(axis=chanDim, epsilon=bnEps,
90                     импульс=bnMom)(x) x =
91         = Activation("relu")(x) x =
92         ZeroPadding2D((1, 1))(x) x =
93         MaxPooling2D((3, 3), strides=(2, 2))( Икс)

```

Строки 79-82 мы уже рассматривали ранее — в этих строках мы применяем один слой 3×3 CONV для CIFAR-10. Однако сейчас мы обновляем архитектуру для проверки наличия Tiny ImageNet (строка 85).

Если мы создаем экземпляр ResNet для Tiny ImageNet, нам нужно добавить несколько дополнительных слоев. Для начала мы применяем слой CONV 5×5 , чтобы изучить более крупные карты объектов (в полной реализации набора данных ImageNet мы фактически будем изучать фильтры 7×7).

Затем мы применяем пакетную нормализацию с последующей активацией ReLU. Максимальное объединение, единственный слой максимального объединения в архитектуре ResNet, применяется к строке 93 с использованием размера 3×3 и шага 2×2 . В сочетании с предыдущим слоем нулевого заполнения (строка 92) объединение гарантирует, что наш выходной пространственный размер тома 32×32 , точно такие же пространственные размеры, как у входных изображений из CIFAR-10. Проверка того, что размер выходного тома составляет 32×32 , гарантирует, что мы можем легко повторно использовать остальную часть реализации ResNet без внесения каких-либо дополнительных изменений.

12.5.2 Обучение ResNet на Tiny ImageNet с помощью метода ctrl + c

Теперь, когда наша реализация ResNet обновлена, давайте напишем скрипт Python, отвечающий за фактический процесс обучения. Откройте новый файл, назовите его train.py и вставьте следующий код:

```

1 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом режиме 2
import matplotlib 3 matplotlib.use("Agg")

```

```

4
5 # импортировать необходимые пакеты 6
из config import tiny_imagenet_config as config 7 из
pyimagesearch.preprocessing import ImageToArrayPreprocessor 8 из
pyimagesearch.preprocessing import SimplePreprocessor 9 из pyimagesearch.preprocessing
import MeanPreprocessor 10 из pyimagesearch.callbacks import EpochCheckpoint 11 из
pyimagesearch.callbacks1 import from Training2Monitor pyimagesearch.io импортировать
HDF5DatasetGenerator 13 из pyimagesearch.nn.conv импортировать ResNet 14 из
keras.preprocessing.image импортировать ImageDataGenerator 15 из keras.optimizers
импортировать SGD 16 из keras.models импортировать load_model 17 импортировать
keras.backend as K 18 импортировать argparse 19 импортировать json 20 систем
импорта

```

21

```

22 # установить высокий предел рекурсии, чтобы Theano не жаловался 23
sys.setrecursionlimit(5000)

```

Строки 2 и 3 настраивают matplotlib, чтобы мы могли сохранять наши цифры и графики на диск во время процесса обучения. Затем мы импортируем остальные наши пакеты Python в строках 6-20. У нас есть

видели все эти импорты раньше из главы 11 на GoogLeNet + Tiny ImageNet, только теперь мы импортируем ResNet (строка 13), а не GoogLeNet. Мы также обновим максимальный предел рекурсии на тот случай, если вы используете серверную часть Theano.

Далее идут наши аргументы командной строки:

```

25 # построить разбор аргумента и разобрать аргументы 26 ap =
argparse.ArgumentParser() 27 ap.add_argument("-c", "--checkpoints",
required=True,
28     help="путь к выходному каталогу контрольных точек")
29 ap.add_argument("-m", "--model", type=str,
30     help="путь к *конкретной* контрольной точке модели для
загрузки") 31 ap.add_argument("-s", "--start-epoch ", тип=целое, по умолчанию=0,
32     help="эпоха, в которой нужно возобновить
обучение") 33 args = vars(ap.parse_args())

```

Эти переключатели командной строки можно использовать для запуска обучения с нуля или перезапуска обучения с определенной эпохи. Более подробный обзор каждого аргумента командной строки см. в разделе 12.3.1.

Давайте также инициализируем наш `ImageDataGenerator`, который будет применяться к обучающему набору для увеличения данных:

```

35 # построить генератор обучающих изображений для увеличения данных
36 aug = ImageDataGenerator(rotation_range=18, zoom_range=0.15,
37     width_shift_range = 0.2, height_shift_range = 0.2, shear_range = 0.15,
38     horizontal_flip = True, fill_mode = «ближайший»)
39
40 # загрузить средства RGB для тренировочного
набора 41 означает = json.loads(open(config.DATASET_MEAN).read())

```

Строка 41 затем загружает наши средние значения RGB, вычисленные по тренировочному набору вычитания средних.

Наши препроцессоры изображений здесь довольно стандартны и состоят из обеспечения изменения размера изображения до 64×64 пикселей, выполнения средней нормализации и последующего преобразования изображения в массив, совместимый с Keras:

```

43 # инициализируем препроцессоры
изображений 44 sp = SimplePreprocessor(64,
64) 45 mp = MeanPreprocessor(means["R"], означает["G"], означает["B"])
46 iap = ImageToArrayPreprocessor()
47
48 # инициализируем генераторы обучающих и проверочных наборов
данных
50
51 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
52     препроцессоры=[sp, mp, iap], классы=config.NUM_CLASSES)

```

Затем на основе препроцессоров изображений и увеличения данных мы можем создать `HDF5DatasetGenerator`. для наборов данных для обучения и проверки (строки 49-52). Пакеты из 64 изображений будут одновременно опрашиваться для этих генераторов и передаваться по сети.

Если мы обучаем ResNet с самой первой эпохи, нам нужно создать экземпляр модели и скомпилировать его:

Наконец, мы начнем процесс обучения и обучим нашу сеть, используя мини-пакеты размером 64:

83 # обучаем сеть 84

```

model.fit_generator(trainGen.generator(),
86      steps_per_epoch=trainGen.numImages // 64,
87      validation_data=valGen.generator(),
88      validation_steps=valGen.numImages // 64, epochs =50,
89      max_queue_size=64 * 2, обратные вызовы = обратные
90      вызовы, подробный = 1)
91
92
93 # закрыть базы
94 trainGen.close() 95 valGen.close()

```

Точное количество эпох, которое нам потребуется для обучения ResNet, на данный момент неизвестно, поэтому мы установим эпохи до большого числа и настроить по мере необходимости.

ResNet на Tiny ImageNet: эксперимент №1

Чтобы начать обучение, я выполнил следующую команду:

\$ python train.py --вывод контрольных точек/контрольные точки

После наблюдения за обучением в течение первых 25 эпох стало ясно, что потери при обучении начинают немного стагнировать (рис. 12.9, вверху слева). Чтобы бороться с этим застаем, я прекратил тренировки, снизил скорость обучения с 1e-1 до 1e-2 и возобновил тренировки:

\$ python train.py --checkpoints вывод/контрольные точки \
--model output/checkpoints/epoch_25.hdf5 --start-epoch 25

Обучение продолжалось с 25-й по 35-ю эпохи с более низкой скоростью обучения. Мы сразу видим преимущество снижения скорости обучения на порядок — потери резко падают, а точность приятно растет. (Рисунок 12.9, вверху справа).

Однако после этого первоначального скачка потери при обучении продолжали снижаться гораздо быстрее, чем потери при проверке. Я снова прекратил обучение на 35-й эпохе, снизил скорость обучения с 1e-2 до 1e-3 и возобновил обучение:

\$ python train.py --checkpoints вывод/контрольные точки \
--model output/checkpoints/epoch_35.hdf5 --start-epoch 35

Я позволил ResNet обучаться еще 5 эпох, так как начал замечать явные признаки переобучения (рис. 12.9, внизу). Потери немного падают при изменении 1e - 3, затем начинают расти, в то время как потери при обучении уменьшаются с большей скоростью — это явный признак переобучения. На этом этапе я полностью прекратил обучение и отметил, что точность проверки составляет 53,14%.

Чтобы определить точность на тестовом наборе, я выполнил следующую команду (сохраняя обратите внимание, что скрипт rank_accuracy.py идентичен скрипту из главы 11 на GoogLeNet:

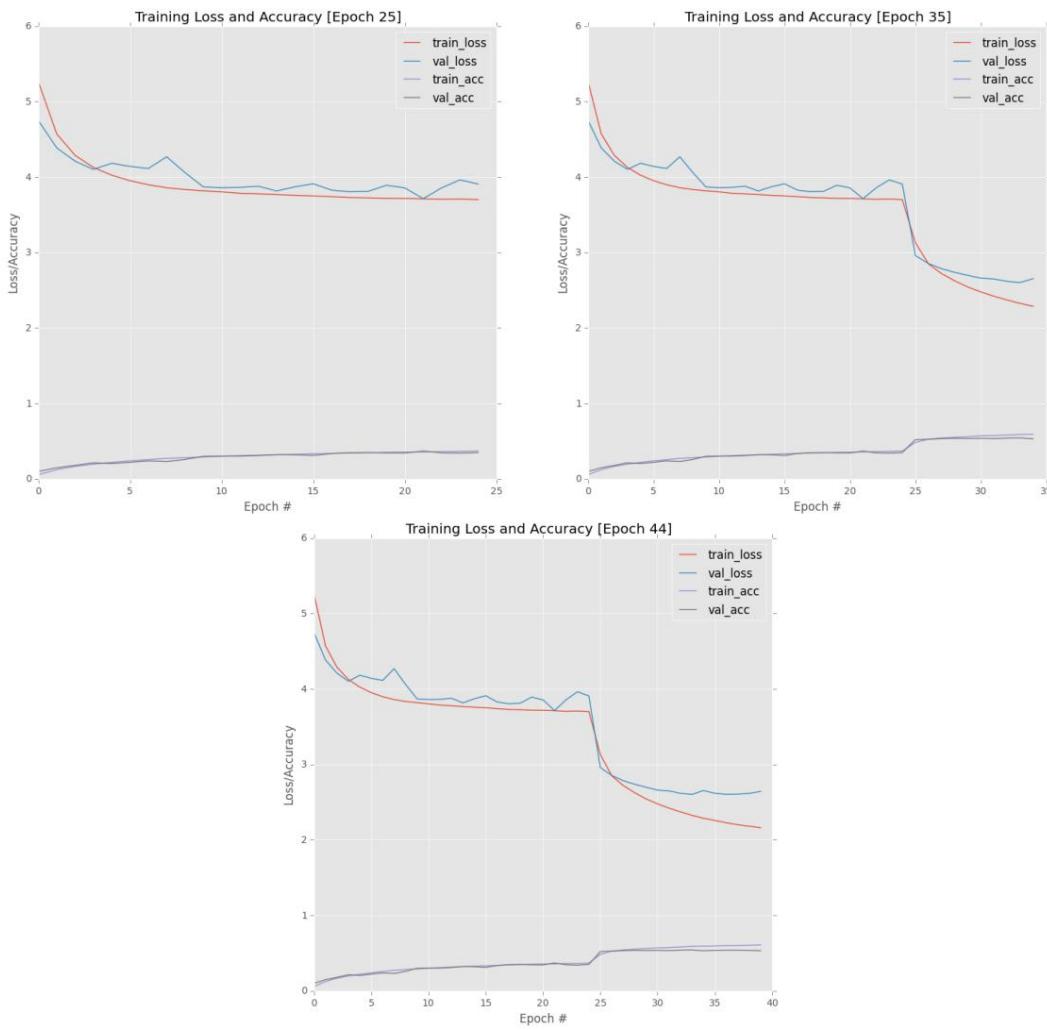


Рисунок 12.9: Вверху слева: первые 25 эпох при обучении ResNet на Tiny ImageNet в эксперименте №1. Вверху справа: следующие 10 эпох. Внизу: последние 5 эпох. Обратите внимание на явный признак переобучения, поскольку потери при проверке начинают увеличиваться в последние эпохи.

```
$ python rank_accuracy.py [INFO]
модель загружена...
[INFO] прогнозирование на основе тестовых данных...
[ИНФО] ранг-1: 53,10%
[ИНФО] ранг-5: 75,43%
```

Как видно из выходных данных, мы получаем 53,10% точности ранга 1 на тестовом наборе. Этот первый эксперимент был неплохим, так как мы уже приближаемся к точности GoogLeNet + Tiny ImageNet . Учитывая успех применения снижения скорости обучения к Tiny ImageNet, я сразу же решил применить тот же процесс к ResNet.

12.5.3 Обучение ResNet на Tiny ImageNet с уменьшением скорости обучения

Чтобы обучить ResNet с использованием снижения скорости обучения, откройте новый файл, назовите его `train_decay.py` и вставьте следующий код:

```

1 # установить бэкэнд matplotlib, чтобы цифры можно было сохранять в фоновом режиме 2
import matplotlib 3 matplotlib.use("Agg")

4
5 # импортировать необходимые пакеты 6
из config import tiny_imagenet_config as config 7 из
pyimagesearch.preprocessing import ImageToArrayPreprocessor 8 из
pyimagesearch.preprocessing import SimplePreprocessor 9 из pyimagesearch.preprocessing
import MeanPreprocessor 10 из pyimagesearch.callbacks import TrainingMonitor 11 из
pyimagesearch.io import HDF5DatasetGenerator pyimagesearch.nn.conv импортировать
ResNet 13 из keras.preprocessing.image импортировать ImageDataGenerator 14 из
keras.callbacks импортировать LearningRateScheduler 15 из keras.optimizers
импортировать SGD 16 импортировать argparse 17 импортировать json 18 импортировать
sys 19 импортировать os

```

```

20
21 # установить высокий предел рекурсии, чтобы Theano не жаловался 22
sys.setrecursionlimit(5000)

```

Строки 6-19 импортируют необходимые пакеты Python. Нам нужно импортировать LearningRateScheduler class в строке 14 , чтобы мы могли применить график скорости обучения к процессу обучения. Ограничение системной рекурсии затем устанавливается в строке 22 на тот случай, если вы используете серверную часть Theano.

Затем мы определяем фактическую функцию, ответственную за применение затухания скорости обучения:

```

24 # определить общее количество эпох для обучения вместе с 25 # начальной
скоростью обучения
26 NUM_EPOCHS = 75
27 INIT_LR = 1e-1

28
29 по определению poly_decay (эпоха):
30     # инициализируем максимальное количество эпох, базовую скорость обучения,
31     # и мощность полинома maxEpochs = NUM_EPOCHS baseLR = INIT_LR power = 1.0
32
33
34
35
36     # вычислить новую скорость обучения на основе полиномиального
37     # распада alpha = baseLR * (1 - (epoch / float(maxEpochs))) ** мощность
38
39     # вернуть новую скорость обучения
40     return alpha

```

Здесь мы указываем, что будем обучаться максимум 75 эпох (строка 26) с базовой скоростью обучения $1e^{-1}$ (строка 27). Функция poly_decay определена в строке 29 , которая принимает единственный параметр — номер текущей эпохи . Мы устанавливаем power=1.0 в строке 34 , чтобы превратить полиномиальное затухание в линейное (подробности см. в главе 11). Новая скорость обучения (на основе текущей эпохи) вычисляется в строке 37, а затем возвращается в вызывающую функцию в строке 40.

Перейдем к аргументам командной строки:

```

42 # построить аргумент parse и разобрать аргументы
43 ap = argparse.ArgumentParser()
44 ap.add_argument("-m", "--model", required=True,
45                 help="путь к выходной модели")
46 ap.add_argument("-o", "--output", required=True,
47                 help="путь к выходному каталогу (журналы, графики и т. д.)")
48 аргументов = вары (ap.parse_args())

```

Здесь нам просто нужно указать путь к нашей сериализованной выходной модели --model после обучения . вместе с -output путем для хранения любых графиков/журналов.

Теперь мы можем инициализировать наш класс увеличения данных и загрузить средства RGB с диска:

```

50 # построить генератор обучающих изображений для увеличения данных
51 августа = ImageDataGenerator (rotation_range = 18, zoom_range = 0,15,
52     width_shift_range=0,2, height_shift_range=0,2, shear_range=0,15,
53     horizontal_flip = Истина, fill_mode = "ближайший")
54
55 # загрузить средства RGB для тренировочного набора
56 означает = json.loads(open(config.DATASET_MEAN).read())

```

А также создадим наши обучающие и проверочные HDF5DatasetGenerators:

```

58 # инициализируем препроцессоры изображений
59 sp = Простой препроцессор (64, 64)
60 mp = MeanPreprocessor (означает ["R"], означает ["G"], означает ["B"])
61 iap = ImageToArrayPreprocessor()
62
63 # инициализировать генераторы наборов данных для обучения и проверки
64 trainGen = HDF5DatasetGenerator(config.TRAIN_HDF5, 64, aug=aug,
65     препроцессоры=[sp, mp, iap], классы=config.NUM_CLASSES)
66 valGen = HDF5DatasetGenerator(config.VAL_HDF5, 64,
67     препроцессоры=[sp, mp, iap], классы=config.NUM_CLASSES)

```

Наш список обратных вызовов будет состоять из TrainingMonitor и LearningRateScheduler:

```

69 # построить набор коллбэков
70 figPath = os.path.sep.join([args["output"], "{}.png".format(
71     os.getpid())])
72 jsonPath = os.path.sep.join([args["output"], "{}.json".format(
73     os.getpid())])
74 обратных вызова = [TrainingMonitor (figPath, jsonPath = jsonPath),
75     LearningRateScheduler (poly_decay)]

```

Ниже мы создаем нашу архитектуру ResNet и оптимизатор SGD, используя те же параметры. как наш первый эксперимент:

```

77 # инициализируем оптимизатор и модель (ResNet-56)
78 print("[INFO] компилируется модель...")
79 модель = ResNet.build(64, 64, 3, config.NUM_CLASSES, (3, 4, 6),
80     (64, 128, 256, 512), reg=0,0005, набор данных="tiny_imagenet")

```

```
81 opt = SGD (lr = INIT_LR, импульс = 0,9) 82
model.compile (loss = "categorical_crossentropy", оптимизатор = opt,
83     метрики=["точность"])
```

Наконец, мы можем обучать нашу сеть мини-пакетами по 64:

```
85 # обучаем сеть
86 print("[INFO] обучающая сеть...") 87
model.fit_generator(trainGen.generator(),
88     steps_per_epoch=trainGen.numImages //
89     64, validation_data=valGen.generator(),
90     validation_steps=valGen.numImages // 64 , эпохи =
91     NUM_EPOCHS, max_queue_size = 64 * 2, обратные
92     вызовы = обратные вызовы, подробный = 1)
93
94
```

Количество эпох, для которых мы собираемся тренироваться, контролируется NUM_EPOCHS , определенным ранее в этом скрипте. Мы будем линейно снижать скорость обучения с 1e-1 до нуля в течение NUM_EPOCHS. И, наконец, сериализуйте модель на диск после завершения обучения:

```
96 # сохранить сеть на диск
97 print("[INFO] сериализующая сеть...") 98
model.save(args["model"])
99
100 # закрыть базы данных
101 trainGen.close() 102
valGen.close()
```

ResNet на Tiny ImageNet: эксперимент № 2 В

в этом эксперименте я обучил описанную выше архитектуру ResNet на наборе данных Tiny ImageNet с использованием оптимизатора SGD, базовой скорости обучения 1e - 1 и импульса 0,9. Скорость обучения линейно уменьшалась в течение 75 эпох. Чтобы выполнить этот эксперимент, я выполнил следующую команду:

```
$ python train_decay.py --model output/resnet_tinyimagenet_decay.hdf5 \
--выход вывод
```

Результирующий график обучения показан на рис. 12.11. Здесь мы можем увидеть драматический эффект , который снижение скорости обучения может оказать на процесс обучения. Хотя и потери при обучении, и потери при проверке отличаются друг от друга, только в эпоху 60 разрыв увеличивается по сравнению с предыдущими эпохами. Кроме того, наши потери при проверке также продолжают уменьшаться. В конце 75-й эпохи я получил 58,32% точности первого ранга.

Затем я оценил сеть на тестовом наборе, используя следующую команду:

```
$ python rank_accuracy.py [INFO]
модель загрузки...
[INFO] прогнозирование на основе тестовых данных...
[ИНФО] ранг-1: 58,03%
[ИНФО] ранг-5: 80,46%
```

ResNet claims the #5 position on the Tiny ImageNet leaderboard with an error of $1 - 0.5803 = 0.4197$, the best result when training from scratch (and not using feature extraction/fine tuning)

Leaderboard

Feature extraction/fine tuning methods			
#	Name	Error Rate	# Submissions
1	Avati,Anand	0.268	14
2	Kim,Hansohl Elliott	0.311	17
3	Qian,Junyang	0.338	6
4	Liu,Fei	0.339	8
5	Zhai,Andrew Huan	0.446	4
6	Shen,William	0.452	9
7	Shcherbina,Anna	0.506	15
8	Ebrahimi,Mohammad Sadegh	0.561	5
9	Ting,Jason Ming	0.616	17
10	Random Guessers	0.995	17
11	Khosla,Vani	0.995	4

Рисунок 12.10: Используя ResNet, мы можем достичь 5-го места в таблице лидеров Tiny ImageNet, обойдя все другие подходы, которые пытались обучить сеть с нуля. Во всех результатах с меньшей ошибкой, чем в нашем подходе, применялась тонкая настройка/извлечение признаков.

Как показали результаты, мы достигли 58,03% точности ранга 1 и 80,46% точности ранга 5 на тестовом наборе, что является существенным улучшением по сравнению с нашей предыдущей главой о GoogLeNet. Этот результат приводит к ошибке теста $1 - 0.5803 = 0.4197$, что легко занимает пятую позицию в таблице лидеров Tiny ImageNet (рис. 12.10).

Это наилучшая точность, полученная в наборе данных Tiny ImageNet, который не выполняет какую-либо форму трансферного обучения или точной настройки. Учитывая, что точность для позиций 1-4 была получена в результате тонкой настройки сети, которая уже была обучена на полном наборе данных ImageNet, трудно сравнивать точно настроенную сеть с сетью, обученной полностью с нуля. Если мы (справедливо) сравним нашу сеть, которая была обучена с нуля, с другими сетями, обученными с нуля в таблице лидеров, мы увидим, что наша ResNet получила самую высокую точность среди группы.

12.6 Резюме

В этой главе мы подробно обсудили архитектуру ResNet, включая микроархитектуру остаточного модуля. Оригинальный остаточный модуль, предложенный He et al. в их статье 2015 года «Глубокое остаточное обучение для распознавания изображений» [24] претерпело множество изменений. Первоначально модуль состоял из двух слоев CONV и «ярлыка» сопоставления идентификаторов. В той же статье было обнаружено, что добавление «узкого места» последовательности слоев CONV 1×1 , 3×3 и 1×1 повысило точность.

Затем в их исследовании 2016 года Identity Mappings in Deep Residual Networks [33] был представлен остаточный модуль перед активацией. Мы называем это обновление «предварительной активацией», потому что мы применяем активацию и нормализацию пакетов до свертки, что противоречит «общепринятым мнениям» при построении сверточных нейронных сетей.

Оттуда мы реализовали архитектуру ResNet, используя как узкое место, так и предварительную активацию с использованием платформы Keras. Затем эта реализация использовалась для обучения ResNet на наборах данных CIFAR-10 и Tiny ImageNet. В CIFAR-10 мы смогли воспроизвести результаты He et al., получив точность 93,58%. Затем в Tiny ImageNet была достигнута точность 58,03%, что является самой высокой точностью для сети, обученной с нуля на Стенфордском cs231n Tiny ImageNet.



Рисунок 12.11: Применение скорости обучения к ResNet при обучении на Tiny ImageNet приводит к точности проверки 58,32% и точности тестирования 58,03%, что значительно выше, чем в нашем предыдущем эксперименте с обучением $\text{ctrl} + \text{c}$.

ВЫЗОВ.

Позже в ImageNet Bundle мы исследуем ResNet и обучим его на полном наборе данных ImageNet, еще раз повторив работу He et al.

Список используемой литературы

- [1] 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. 11, 85, 86).
- [2] Джейфри Хинтон. Нейронные сети для машинного обучения. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (цитируется на стр. 11, 85).
- [3] Команда Kaggle. Kaggle: Собаки против кошек. <https://www.kaggle.com/c/dogs-vs-cats> (цитируется на стр. 12, 95).
- [4] Андрей Карпаты. Крошечный вызов ImageNet. <http://cs231n.stanford.edu/project.html> (цитируется на стр. 12, 131, 168).
- [5] Ян Гудфеллоу, Йошуа Бенджио и Аарон Курвиль. Глубокое обучение. <http://www.deeplearningbook.org>. MIT Press, 2016 (цитируется на стр. 13, 85, 86).
- [6] Алекс Крижевский, Илья Суцквер и Джейфри Э. 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. 14, 86, 95, 103, 128).
- [7] Янн Лекун и др. «Градиентное обучение для распознавания документов». В: Продолжить IEEE. 1998, страницы 2278–2324 (цитируется на странице 14).
- [8] Адриан Роузброк. Гуру PyImageSearch. <https://www.pyimagesearch.com/pyimagesearch-guru/>. 2016 (цитируется на стр. 16).
- [9] 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 стр. 16).

- [10] 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.unitrier.de/db/conf/cvpr/cvpr2006-2.html#NilsbackZ06> (цитируется на стр. 17).
- [11] 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. 32, 86, 171).
- [12] Группа HDF. Версия 5 формата иерархических данных. <http://www.hdfgroup.org/HDF5> (цитируется на стр. 33).
- [13] Эндрю Нг. Машинное обучение. <https://www.coursera.org/learn/machine-learning> (цитируется на стр. 44).
- [14] 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-адрес: <http://dx.doi.org/10.1109/CVPR.2005.177>. (цитируется на стр. 47, 57).
- [15] 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. 47, 57).
- [16] 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 47, 57).
- [17] Кристиан Сегеди и др. «Погружаемся глубже с извилинами». В: Компьютерное зрение и распознавание образов (CVPR). 2015. URL: <http://arxiv.org/abs/1409.4842>. (цитируется на стр. 51, 81, 86, 131, 133, 168).
- [18] Йоав Фройнд и Роберт Э. Шапире. «Теоретическое обобщение онлайн-обучения и приложение к повышению». В: J. Comput. Сист. науч. 55.1 (август 1997 г.), страницы 119–139. ISSN: 0022-0000. DOI: 10.1006/jcss.1997.1504. URL-адрес: <http://dx.doi.org/10.1006/jcss.1997.1504> (цитируется на стр. 71).
- [19] 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. 71).
- [20] Л. Брейман и др. Деревья классификации и регрессии. Монтерей, Калифорния: Уодсворт и Брукс, 1984 г. (цитируется на стр. 71).
- [21] Тревор Хести, Роберт Тибширани и Джером Фридман. Элементы статистического обучения. Серия Спрингера по статистике. Нью-Йорк, штат Нью-Йорк, США: Springer New York Inc., 2001 (цитируется на стр. 71).
- [22] Куонг Нгуен, Йонг Ван и Ха Нам Нгуен. «Классификатор случайного леса в сочетании с выбором признаков для диагностики и прогнозирования рака молочной железы». В: Журнал биомедицинских наук и техники (2013). URL: http://file.scirp.org/Html/6-9101686_31887.htm (цитируется на стр. 72).

- [23] Томас Г. Диттерих. «Методы ансамбля в машинном обучении». В: Материалы Первого международного семинара по множественным системам классификаторов. МКС '00. Лондон, Великобритания, Великобритания: Springer-Verlag, 2000, страницы 1–15. ISBN: 3-540-67704-6. URL: <http://dl.acm.org/citation.cfm?id=648054.743935> (цитируется на стр. 72).
- [24] Кайминг Хе и др. «Глубокое остаточное обучение для распознавания изображений». В: CoRR abs/1512.03385 (2015). URL: <http://arxiv.org/abs/1512.03385> (цитируется на стр. 81, 131, 171, 192, 196, 202).
- [25] Гао Хуанг и др. «Ансамбли снимков: Поезд 1, получи М бесплатно». В: CoRR abs/1704.00109 (2017). URL: <http://arxiv.org/abs/1704.00109> (цитируется на стр. 81).
- [26] Андрей Карпаты. Нейронные сети (часть III). <http://cs231n.github.io/neural-networks-3/> (цит. на стр. 83, 85).
- [27] Себастьян Рудер. «Обзор алгоритмов оптимизации градиентного спуска». В: CoRR abs/1609.04747 (2016). URL: <http://arxiv.org/abs/1609.04747> (цитируется на стр. 83).
- [28] Джон Дучи, Элад Хазан и Йорам Сингер. «Адаптивные субградиентные методы для онлайн -обучения и стохастической оптимизации». В: Дж. Мах. Учиться. Рез. 12 (июль 2011 г.), страницы 2121–2159 . ISSN: 1532-4435 . URL-адрес: <http://dl.acm.org/citation.cfm?doid=1953048.2021068> (цитируется на стр. 84).
- [29] Мэтью Д. Зейлер. «ADADELTA: метод адаптивной скорости обучения». В: CoRR abs/1212.5701 (2012). URL: <http://arxiv.org/abs/1212.5701> (цитируется на стр. 84).
- [30] Тимоти Дозат. Включение Нестерова Импульса в Адама. http://cs229.stanford.edu/proj2015/054_report.pdf (цитируется на стр. 86).
- [31] Том Шаул, Иоаннис Антоноглу и Дэвид Сильвер. «Юнит-тесты для стохастической оптимизации». В: CoRR abs/1312.6055 (2013). URL: <http://arxiv.org/abs/1312.6055> (цитируется на стр. 86).
- [32] Форрест Н. Яндоля и др. «SqueezeNet: точность уровня AlexNet с в 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. 86, 131).
- [33] Кайминг Хе и др. «Отображение идентичности в глубоких остаточных сетях». В: CoRR abs/1603.05027 (2016). URL: <http://arxiv.org/abs/1603.05027> (цитируется на стр. 86, 172, 174, 202).
- [34] Эндрю Нг. Основы создания приложений с использованием глубокого обучения. <https://nips.cc/Conferences/2016/Schedule?showEvent=6203>. 2016 (цитируется на стр. 89, 91).
- [35] Томаш Малисевич. Основы создания приложений глубокого обучения: Ng на NIPS2016. <http://www.blog.computer-vision.com/2016/12/nuts-and-bolts-of-building-deep.html> (цитируется на стр. 89).
- [36] Андрей Карпаты. Трансферное обучение. <http://cs231n.github.io/transfer-learning/> (цитируется на стр. 93).
- [37] Грег Чу. Как использовать трансферное обучение и тонкую настройку в Keras и Tensorflow для создания систему распознавания изображений и классифицировать (почти) любой объект. <https://deeplearningsandbox.com/how-to-use-transfer-learning-and-dine-tuning-in-keras-and-tensorflow-to-build-an-image-recognition-94b0b02444f2> (цитируется на стр. 93).
- [38] Адриан Роузброк. Практические Python и OpenCV + тематические исследования. PyImageSearch.com, 2016. URL: <https://www.pyimagesearch.com/practical-python-opencv/> (цитируется на стр. 105).
- [39] Андрей Карпаты. CS231n: сверточные нейронные сети для визуального распознавания. <http://cs231n.stanford.edu/>. 2016 (цитируется на стр. 131, 146).

- [40] Мин Линь, Цян Чен и Шуичэн Ян. «Сеть в сети». В: CoRR абс/1312.4400 (2013). URL: <http://arxiv.org/abs/1312.4400> (цитируется на стр. 132).
- [41] Йост Тобиас Спрингенберг и др. «Стремление к простоте: полностью сверточная сеть». В: CoRR абс/1412.6806 (2014). URL: <http://arxiv.org/abs/1412.6806> (цитируется по стр. 133, 171).
- [42] Чиюань Чжан и др. «Понимание глубокого обучения требует переосмысления обобщения». В: CoRR абс/1611.03530 (2016). URL: <http://arxiv.org/abs/1611.03530> (цитируется по стр. 133).
- [43] Ворднет. О Ворднет. <http://wordnet.princeton.edu>. 2010 (цитируется на стр. 147).
- [44] Ксавье Глорот и Йошуа Бенжио. «Понимание сложности обучения глубокой упреждающей нейронные сети». В: Материалы Международной конференции по искусственному интеллекту. и статистика (АИСТАС'10). Общество искусственного интеллекта и статистики. 2010 (цитируется по стр. 171).
- [45] Кайминг Хе и др. «Углубление в выпрямители: превосходящие человеческие возможности на уровне Классификация ImageNet». В: CoRR абс/1502.01852 (2015). URL-адрес: <http://arxiv.org/abs/1502.01852> (цитируется на стр. 171).
- [46] Каймин Хэ. Глубокие остаточные сети. <https://github.com/KaimingHe/> глубокий - остаточные сети (цитируется на стр. 176).
- [47] Вэй Ву. Реснет. <https://github.com/tornadomeet/ResNet> (цитируется на стр. 176).
- [48] Каймин Хэ. ResNet: должны ли слои свертки иметь смещения? <https://github.com/KaimingHe/deep-residual-networks/issues/10#issuecomment-194037195> (цитируется на странице 177).
- [49] Родриго Бененсон. CIFAR-10: Кто лучший в CIFAR-10? http://rodrijob.github.io/_ мы _ там _ еще/сборка/классификация _ наборы данных _ полученные результаты . HTML # 43494641522d3130 (цитируется на стр. 180, 192).
- [50] Сообщество Теано. Theano: максимальный предел рекурсии. <https://github.com/Theano/issues/689> (цитируется на стр. 181).