O'REILLY®

Программируем с PyTorch

Создание приложений глубокого обучения

Ян Пойнтер

Programming PyTorch for Deep Learning

Creating and Deploying Deep Learning Applications

Ian Pointer

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Программируем с PyTorch

Создание приложений глубокого обучения

Ян Пойнтер

ПИТЕР®

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

2020

ББК 32.973.2-018.1
УДК 004.43
П47

Пойнтер Ян

П47 Программируем с PyTorch: Создание приложений глубокого обучения. — СПб.: Питер, 2020. — 256 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1677-5

PyTorch — это фреймворк от Facebook с открытым исходным кодом. Узнайте, как использовать его для создания собственных нейронных сетей.

Ян Пойнтер поможет разобраться, как настроить PyTorch в облачной среде, как создавать нейронные архитектуры, облегчающие работу с изображениями, звуком и текстом. Книга охватывает важнейшие концепции применения переноса обучения, модели отладки и использования библиотеки PyTorch.

Вы научитесь:

- Внедрять модели глубокого обучения в работу.
- Использовать PyTorch в масштабных проектах.
- Применять перенос обучения.
- Использовать PyTorch torchaudio и сверточные модели для классификации аудиоданных.
- Применять самые современные методы NLP, используя модель, обученную на «Википедии».
- Выполнять отладку моделей PyTorch с TensorBoard и флеймграф.
- Развертывать приложения PyTorch в контейнерах.

«PyTorch — это одна из самых быстрорастущих библиотек глубокого обучения, соперничающая с гигантом Google — TensorFlow — практически на равных. Книга обязательно должна стать настольной для каждого программиста и разработчика алгоритмов машинного обучения, которые хотят использовать PyTorch в своей работе».

Анкур Патель, вице-президент направления Data Science в компании 7Park Data.

Ян Пойнтер (Ian Pointer) — дата-инженер, создает решения машинного обучения для клиентов из списка Fortune 100. В настоящее время работает в Lucidworks, где занимается разработкой NLP-приложений и проектированием.

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

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1492045359 англ.
Authorized Russian translation of the English edition of Programming PyTorch for Deep Learning
ISBN 9781492045359 © 2019 Ian Pointer
This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.
ISBN 978-5-4461-1677-5

КРАТКОЕ СОДЕРЖАНИЕ

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Глубокое обучение в современном мире

Всем привет! В этой книге я расскажу вам о глубоком обучении в PyTorch — библиотеке с открытым исходным кодом, выпущенной Facebook в 2017 году. Если только вы не живете на необитаемом острове, то наверняка заметили, что нейронные сети сейчас можно встретить повсюду. Они перестали быть некоей суперчастью computer science, которую мы изучаем и никак не используем: нейросети есть, например, в наших смартфонах. С их помощью мы улучшаем фотографии или даем голосовые команды. Программное обеспечение для электронной почты читает письма и выдает соответствующий контексту ответ, нас слушаются умные колонки, автомобили ездят сами по себе, а компьютер наконец-то переиграл человека в Го. А, например, в авторитарных странах эта технология используется для не самых хороших дел: системы, в основе которых лежат нейронные сети, распознают лица и принимают решения о задержании этих людей.

И все же, несмотря на ощущение, что все произошло так быстро, концепции глубокого обучения и нейронных сетей уходят в далекое прошлое. Доказательство того, что такая сеть может стать заменой *любой* математической функции аппроксимации (нейронные сети могут быть обучены для множества различных задач), датируется 1989 годом[1], а в конце 90-х сверточные нейронные сети использовались для распознавания контрольных цифр. Все это время строился прочный фундамент, но почему же именно за последние десять лет произошел бум?

[1] См. «Approximation by Superpositions of Sigmoidal Functions», George Cybenko (1989), https://www.semanticscholar.org/paper/Approximation-by-superpositions-of-a-sigmoidal-Cybenko/8da1dda34ecc96263102181448c94ec7d645d085

Существует много причин, но основная из них заключается в росте производительности графических процессоров (GPU) и их ценовой доступности.

Изначально разработанные для игр, GPU должны выполнять миллионы операций над матрицами в секунду, чтобы визуализировать все полигоны гоночной игры или стрелялки, в которую вы играете на консоли или на ПК, — операции, под которые стандартный центральный процессор (CPU) просто не заточен.

В работе «Large-Scale Deep Unsupervised Learning Using Graphics Processors», которую Раджат Райна написал совместно с другими авторами, отмечается, что обучение нейронных сетей также основано на выполнении множества операций над матрицами, и поэтому дополнительные видеокарты могут использоваться для ускорения обучения, а также для создания более крупных и *глубоких* архитектур нейронных сетей.

Другие важные методы, такие как прореживание, или дропаут (который мы рассмотрим в главе 3), также были введены в последнее десятилетие как способ не только ускорить обучение, но и сделать его более *обобщенным* (чтобы сеть не просто научилась распознавать набор данных для обучения, как это происходит при возникновении проблемы переобучения, о которой мы поговорим в главе 1). За последние пару лет компании вывели этот подход, в основе которого лежат GPU, на новый уровень, и Google создал так называемые тензорные процессоры (TPU), которые представляют собой устройства, разработанные специально для максимально быстрого глубокого обучения. Они доступны как часть экосистемы Google Cloud.

Еще один способ оценить прогресс в области глубокого обучения за последнее десятилетие — это конкурс ImageNet. Обширная база данных, насчитывающая более 14 миллионов изображений, вручную разделенных на 20 тысяч категорий, ImageNet — сокровищница размеченных данных для целей машинного обучения.

С 2010 года ведется проект ImageNet Large Scale Visual Recognition Challenge — кампания по широкомасштабному распознаванию образов

в ImageNet, в рамках которой различные программные продукты ежегодно соревнуются в классификации и распознавании объектов и сцен в базе данных из тысячи категорий ImageNet, и до 2012 года процент ошибок составлял около 25 %.

Однако в тот год победила глубокая сверточная нейронная сеть, значительно превзошедшая всех остальных участников: ее процент ошибок составлял 16 %. В последующие годы уровень ошибок все больше и больше снижался, и в 2015 году архитектура ResNet получила результат 3,6 %, что превосходит средний результат человека в ImageNet (5 %). Нас обошли.

Что такое глубокое обучение и нужна ли докторская степень, чтобы понять его?

Определение глубокого обучения несколько сложнее, чем кажется. По одному из определений глубокое обучение — это техника машинного обучения, в которой используются многочисленные и множественные слои нелинейных преобразований для постепенного извлечения признаков из необработанных входных данных. Безусловно, так и есть, но понятнее не становится, правда? Я предпочитаю описывать глубокое обучение как метод решения задач, предоставляющий входные данные и необходимые выходные данные и позволяющий компьютеру найти решение, используя нейронную сеть.

Математика — вот что пугает многих, если речь идет о глубоком обучении. Посмотрите любую работу в этой области, и вы увидите огромное количество греческих букв. Скорее всего, вы пуститесь наутек без оглядки. Но на самом деле не нужно быть гением, чтобы использовать методы глубокого обучения. Для большинства повседневных базовых применений технологии не нужно много знать, а чтобы по-настоящему понять ее (как вы увидите в главе 2), нужно всего лишь немного подготовиться. Это поможет понять концепции, которые вы наверняка учили в старших классах. Так что не пугайтесь математики.

К концу главы 3 вы сможете всего за несколько строк кода построить классификатор изображений, который составит достойную конкуренцию предложенным лучшими умами 2015 года.

PyTorch

Как я уже говорил, PyTorch — это предложение от Facebook с открытым исходным кодом, которое облегчает написание кода для глубокого изучения на Python. У него два «родителя». Во-первых, что вовсе не удивительно, учитывая его название, много функций и концепций он позаимствовал из Torch — библиотеки нейронных сетей на основе Lua, появившейся в 2002 году. Другой «родитель» — Chainer, разработанный в Японии в 2015 году. Chainer была одной из первых библиотек, предложивших энергичный подход к дифференциации вместо определения статических графов, позволяющий более гибко подходить к созданию, обучению и эксплуатации сетей. Наследие Torch и идеи Chainer сделали PyTorch популярным за последние пару лет.[1]

Библиотека также содержит модули, которые помогают работать с текстом, изображениями и звуком (`torchtext`, `torchvision` и `torchaudio`), а также встроенные варианты распространенных архитектур, таких как ResNet (с весами, которые можно загрузить для облегчения работы с *переносом обучения*, о котором вы узнаете в главе 4).

Так же как и Facebook, PyTorch быстро завоевал признание, и такие компании, как Twitter, Salesforce, Uber и NVIDIA, используют его различными способами для глубокого обучения. Я знаю, что вы хотите спросить...

А как насчет TensorFlow?

Да, давайте посмотрим на весьма заметного слона от Google. Что такого предлагает PyTorch, чего нет у TensorFlow? Почему вы должны изучать именно PyTorch?

[1] Обратите внимание, что PyTorch заимствует только идеи Chainer, а не реальный код.

Ответ заключается в том, что традиционный TensorFlow работает не так, как PyTorch, что значительно влияет на написание кода и отладку. В TensorFlow используется библиотека для построения представления архитектуры нейронной сети в виде графа, а затем выполняются операции с этим графом в рамках библиотеки TensorFlow. Этот метод декларативного программирования несколько расходится с более императивной парадигмой Python, а это означает, что программы на Python в TensorFlow могут выглядеть странно и быть трудными для понимания. Другая проблема заключается в том, что объявление статического графа может сделать динамическое изменение архитектуры во время обучения и время вывода значительно сложнее и шаблоннее, чем в PyTorch.

Поэтому PyTorch стал популярным в исследовательских сообществах. Количество работ, представленных на Международную конференцию по обучению, в которых упоминается PyTorch, за прошедший год подскочило на 200 %, а количество работ, упоминающих TensorFlow, увеличилось почти в равной степени. PyTorch пришел определенно для того, чтобы остаться.

Однако в более поздних версиях TensorFlow все меняется. Недавно в библиотеку была добавлена новая функция, называемая *энергичным выполнением* (eager execution), которая позволяет ей работать по аналогии с PyTorch и будет представлять парадигму, поддерживаемую в TensorFlow 2.0. А так как новых ресурсов, кроме Google, помогающих изучить этот новый метод работы с TensorFlow, не так уж и много, потребуются годы работы, чтобы понять другую парадигму и получить максимальную отдачу от библиотеки. Но это не повод плохо относиться к TensorFlow; она остается проверенной библиотекой, которую поддерживает одна из крупнейших компаний на планете. Я бы сказал, что в PyTorch (поддерживаемый другой крупнейшей компанией) вложен более оптимизированный и целенаправленный подход к глубокому обучению и дифференциальному программированию. Поскольку ему не нужно продолжать поддерживать более старые и сложные API-интерфейсы, учиться и работать в PyTorch легче, чем в TensorFlow.

А как сюда вписывается Keras? Тоже хороший вопрос! Keras — это библиотека глубокого обучения высокого уровня, которая изначально поддерживала Theano и TensorFlow, а теперь также поддерживает некоторые другие платформы, например Apache MXNet. Она предоставляет определенные функции, такие как циклы обучения, проверки и тестирования,

которые низкоуровневые фреймворки оставляют разработчикам, а также простые методы построения архитектур нейронных сетей. Библиотека Keras внесла огромный вклад в освоение TensorFlow и теперь является ее непосредственной частью (как `tf.keras`), а также продолжает оставаться отдельным проектом. PyTorch — это нечто среднее между низкоуровневыми сырыми TensorFlow и Keras; приходится писать свои собственные процедуры обучения и выводы, но создавать нейронные сети почти так же просто (и я бы сказал, что разработчики Python считают подход PyTorch к созданию и повторному использованию архитектур гораздо более логичным, чем некоторые из чудес Keras).

Хотя PyTorch распространен в исследовательских целях, во время чтения вы увидите, что PyTorch 1.0 идеально подходит для производственных сценариев использования.

Типографские соглашения

В данной книге используются следующие типографские соглашения:

Курсив

Обозначает новые термины.

Шрифт без засечек

Обозначает URL, адреса электронной почты, названия файлов и расширения.

`Моноширинный шрифт`

Используется для примеров программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, переменные среды, операторы и ключевые слова.

Этот элемент означает подсказку или предложение.

Этот элемент означает общее примечание.

Этот элемент указывает на предупреждение или предостережение.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания по адресу https://oreil.ly/pytorch-github.

Эта книга призвана облегчить вашу работу. В целом, если к книге прилагается какой-либо пример кода, вы можете использовать его в ваших программах и документации. Обращаться к нам за разрешением нет необходимости, разве что если вы копируете значительную часть кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует отдельного разрешения. Для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение, конечно, нужно. Ответ на вопрос путем цитирования этой книги и цитирования примеров кода не требует обращения к нам. Включение значительного количества кода примеров из этой книги в документацию к вашему продукту лучше обговорить.

Если вам кажется, что использование вами примеров кода выходит за рамки правомерного применения или данных выше разрешений, не стесняясь связывайтесь с нами по адресу permissions@oreilly.com.

Благодарности

Выражаю огромную благодарность моему редактору Мелиссе Поттер (Melissa Potter), моей семье и Тэмми Эдлунд (Tammy Edlund) за их помощь в создании книги. Спасибо научным редакторам, которые предостав-

ляли ценную обратную связь на протяжении всего процесса написания этой книги, включая Фила Роудса (Phil Rhodes), Дэвида Мерца (David Mertz), Чарльза Гивра (Charles Givre), Доминика Монна (Dominic Monn), Анкура Пателя (Ankur Patel) и Сару Наги (Sarah Nagy).

От издательства

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

Мы будем рады узнать ваше мнение!

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

ГЛАВА 1

Начало работы с PyTorch

В этой главе мы настроим все необходимое для работы с PyTorch, и как только сделаем это, каждая последующая глава будет опираться на первоначальный фундамент, поэтому важно во всем разобраться. Отсюда следует первый фундаментальный вопрос: стоит ли создавать настроенный компьютер для глубокого обучения или просто использовать один из множества доступных облачных ресурсов?

Сборка компьютера для глубокого обучения

При погружении в глубокое обучение возникает потребность создать себе монстра для всех своих вычислительных потребностей. Вы можете целыми днями просматривать различные типы видеокарт, изучать возможности различных ЦП, выбирать лучший тип памяти, который можно купить, и, в конце концов, думать о том, насколько большой SSD-накопитель можно приобрести, чтобы сделать доступ к диску максимально быстрым. Я не говорю, что это лишнее; пару лет назад я потратил месяц на то, чтобы составить список комплектующих и собрать новый компьютер на кухонном столе.

Мой вам совет, особенно если вы новичок: не делайте так. Вы можете запросто потратить несколько тысяч долларов на систему, которую не будете использовать так уж много. Лучше поработайте с этой книгой, используя облачные ресурсы (в Amazon Web Services, Google Cloud или Microsoft Azure), и только тогда начинайте думать о создании собственного компьютера, если почувствуете, что требуется один компьютер для работы 24/7. Не нужно вкладывать огромные средства в аппаратное обеспечение для запуска любого кода из этой книги.

Возможно, вам даже не потребуется собирать себе специальный компьютер. Всегда приятно осознавать, что создание собственной установки может обойтись дешевле, если вы знаете, что расчеты всегда будут ограничены одной машиной (не более чем несколькими GPU).

Однако если вычисления начинают требовать нескольких машин и GPU, облако снова вызывает интерес. Учитывая стоимость сборки специального компьютера, я бы хорошенько подумал, прежде чем взяться за это всерьез.

Если я еще не убедил вас, в следующих разделах вы найдете предложения о том, как создать свою систему.

Графический процессор

Сердце каждого блока глубокого обучения, GPU в видеокарте, служит основой для большинства вычислений в PyTorch, и, скорее всего, это будет самый дорогой компонент компьютера. В последнее время цены на видеокарты выросли, а поставки уменьшились из-за их использования в майнинге криптовалют. К счастью, пузырь уже сдувается, и проблем с покупкой становится меньше.

На момент написания этой книги я рекомендовал бы приобрести видеокарту NVIDIA GeForce RTX 2080 Ti. Если нужен вариант подешевле, то смело выбирайте 1080 Ti (хотя, если решите приобрести 1080 Ti исходя из финансовых соображений, предлагаю все-таки рассмотреть облачные технологии). Несмотря на наличие графических карт AMD, их поддержка в PyTorch в настоящее время недостаточно хороша. Поэтому я рекомендую NVIDIA. Обратите внимание на технологию ROCm, которая должна сделать их надежной альтернативой в области GPU.

Центральный процессор / материнская плата

Возможно, вы захотите купить материнскую плату серии Z370. Многие скажут, что CPU для глубокого обучения не так важен и что можно обойтись более медленным, если есть мощный GPU. Но вы удивитесь, как часто CPU становится проблемой, особенно при работе с аугментированными данными.

Оперативная память

Больше оперативной памяти — это хорошо, так как это означает, что можно хранить больше данных, не обращаясь к гораздо более медленной дисковой системе хранения (это особенно важно на этапах обучения модели). Рассчитывайте на как минимум 64 ГБ памяти DDR4.

Хранилище

Хранилище должно быть разделено на два класса: во-первых, твердотельный накопитель с интерфейсом M2 (SSD) — настолько большой, насколько вы можете себе позволить, чтобы у вас был максимально быстрый доступ к *горячим* данным при активной работе над проектом. Для хранилищ второго класса добавьте диск Serial ATA (SATA) емкостью 4 ТБ для данных, с которыми вы не работаете активно, и переносите их в *горячее* и *холодное* хранилище по мере необходимости.

Советую обратить внимание на PCPartPicker, чтобы понять, какие системы для глубокого обучения используют другие (иногда идеи бывают довольно странными). Вы получите представление о необходимых комплектующих и ценах, которые могут сильно колебаться, особенно если речь идет о GPU.

Теперь, когда вы изучили локальные настройки и опции компьютера, самое время перейти к облачным технологиям.

Глубокое обучение в облаке

Итак, почему же облачный вариант лучше? Особенно если учесть, что схема ценообразования Amazon Web Services (AWS) позволяет окупить систему для машинного обучения в течение шести месяцев. Подумайте: вначале вы не будете использовать компьютер 24/7 в течение этих шести месяцев. Просто не будете. Это значит, что можно отключить облачную машину и платить копейки за хранящиеся в это время данные.

Новичкам не нужно выкладываться по полной и использовать один из гигантов NVIDIA — Tesla V100, подключенную к облаку. Можно начать

с одного из более дешевых (иногда даже бесплатных) вариантов на основе K80 и перейти к более мощной карте, когда возникнет необходимость. Это в несколько раз дешевле, чем покупка базовой видеокарты и обновление до 2080Ti. Кроме того, если хотите добавить восемь карт V100 к недублирующему хранению, это можно сделать всего в несколько кликов. Попробуйте проделать то же самое со своим собственным оборудованием.

Другая проблема — обслуживание. Если вы возьмете за привычку регулярно восстанавливать экземпляры в облаке (в идеале, начинать заново каждый раз, когда возвращаетесь к работе над своими экспериментами), у вас почти всегда будет обновленная система. Если у вас свой компьютер, обновление остается за вами. Признаюсь: у меня есть собственный кастомизированный компьютер для глубокого обучения, а я так долго игнорировал установку Ubuntu, что поддерживаемые обновления сильно устарели. Пришлось потратить целый день, чтобы система вернулась к работе и снова могла обновляться. Какая досада!

В любом случае вы решили перейти в облако. Ура! Далее: какой провайдер?

Облачный сервис Google Colaboratory

Прежде чем мы перейдем к провайдерам, давайте подумаем вот о чем. А что, если вы вообще не хотите делать что-то? Не хотите заниматься этой нудной сборкой компьютера или проходить этапы установки экземпляров в облаке? Где вариант для ленивых? У Google есть отличное решение.

Colaboratory (или *Colab*) (https://colab.research.google.com/) — это в основном бесплатная среда Jupyter Notebook, не требующая установки. Понадобится лишь учетная запись Google, чтобы настроить свои собственные блокноты. На рис. 1.1 показан скриншот блокнота, созданного в Colab.

Colab — отличный способ погрузиться в глубокое обучение, так как включает в себя предустановленные версии TensorFlow и PyTorch. Ничего настраивать не нужно. Только ввести `import torch`, и каждый пользователь может получить бесплатный доступ к NVIDIA T4 GPU до 12 часов непрерывной работы. Бесплатно! Для сравнения, эмпирические исследования показывают, что вы получаете примерно половину скорости 1080 Ti для обучения, но с дополнительными 5 ГБ памяти для хранения больших моделей. За дополнительную плату в Colab можно подключаться к более

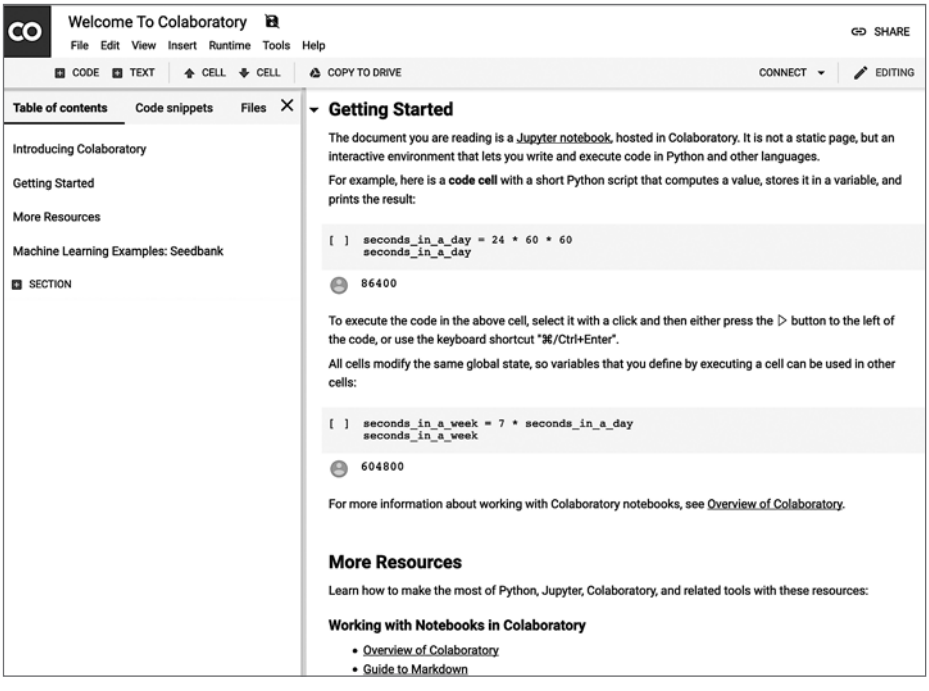

Рис. 1.1. Google Colab(oratory)

мощным GPU и специальному оборудованию тензорного процессора Google, однако все примеры из этой книги можно сделать в Colab бесплатно. Я советую сначала использовать Colab вместе с этой книгой, а затем, при необходимости, перейти к выделенным облачным экземплярам и/или собственному личному серверу для глубокого обучения.

Использование Colab не требует усилий с вашей стороны. Однако если вы хотите больше контроля над системой или доступа к Secure Shell (SSH) — вашему экземпляру в облаке, — то давайте посмотрим, что предлагают поставщики облачных услуг.

Облачные провайдеры

Каждый из трех крупнейших облачных провайдеров (Amazon Web Services, Google Cloud Platform и Microsoft Azure) предлагают экземпляры на основе GPU (также называемые виртуальными машинами, или *VM*)

и официальные образы для развертывания в этих экземплярах. В них есть все необходимое, чтобы начать работу без самостоятельной установки драйверов и библиотек Python. Давайте посмотрим предложения провайдеров.

Amazon Web Services

AWS, 800-фунтовая горилла облачного рынка, с лихвой удовлетворит ваши потребности в GPU и предлагает типы экземпляров P2 и P3. (Тип экземпляра G3, как правило, чаще используется в графических приложениях, например в кодировании видео, поэтому мы не будем его здесь рассматривать.) Экземпляры P2 используют устаревшие карты NVIDIA K80 (максимум 16 могут быть подключены к одному экземпляру), а экземпляры P3 используют невероятно быстрые карты NVIDIA V100 (и вы можете привязать 8 из них к одному экземпляру, если захотите).

Если вы собираетесь использовать AWS, рекомендую взять класс p2.xlarge. Это обойдется вам всего в 90 центов в час и обеспечит достаточную мощность для работы с экземплярами. Возможно, вы захотите перейти к классам P3, когда начнете работать над какими-то сложными заданиями Kaggle.

Создать платформу для глубокого обучения в AWS невероятно просто:

1. Войдите в консоль AWS.
2. Выберите EC2 и нажмите Launch Instance.
3. Найдите параметр Deep Learning AMI (Ubuntu) и выберите его.
4. Выберите p2.xlarge в качестве типа вашего экземпляра.
5. Запустите экземпляр, либо создав новую пару ключей, либо повторно используя существующую пару ключей.
6. Подключитесь к экземпляру, используя SSH и перенаправив порт 8888 на локальном компьютере на экземпляр:

```
ssh -L localhost:8888:localhost:8888 \
-i your .pem filename ubuntu@your instance DNS
```

7. Запустите Jupyter Notebook, введя `jupyter notebook`. Скопируйте сгенерированный URL-адрес и вставьте его в браузер для доступа к Jupyter.

Не забудьте закрыть свой экземпляр, если не используете его! Это можно сделать, кликнув правой кнопкой мыши на экземпляре в веб-интерфейсе и завершив работу. Это закроет экземпляр, и вы таким образом не будете платить за него, пока он не запущен.

Однако *придется* платить за выделенное место его хранения, даже если экземпляр выключен. Имейте это в виду. Чтобы полностью удалить экземпляр и хранилище, выберите параметр завершения.

Azure

Как и AWS, Azure предлагает набор более дешевых экземпляров на основе K80 и более дорогих экземпляров Tesla V100. Azure также использует экземпляры, основанные на более старом программном обеспечении P100, в качестве промежуточной точки между двумя другими. Для целей этой книги рекомендую тип экземпляра, использующий один K80 (NC6), который также стоит 90 центов в час. По мере необходимости можно переходить на другие типы NC, NCv2 (P100) или NCv3 (V100).

Вот как нужно настраивать виртуальную машину в Azure:

1. Войдите на портал Azure и найдите образ Data Science Virtual Machine на Azure Marketplace.

2. Кликните на кнопке Get It Now.

3. Заполните данные виртуальной машины (дайте ей имя, выберите SSD-диск поверх HDD, имя пользователя/пароль SSH, подписку на оплату экземпляров и задайте ближайшее местоположение, которое предлагает тип экземпляра NC).

4. Кликните на кнопке Create option. Экземпляр должен быть готов примерно через пять минут.

5. Можно использовать SSH с именем пользователя/паролем, которые были указаны для публичного имени системы доменных имен (DNS) этого экземпляра.

6. Jupyter Notebook должен запускаться после создания экземпляра; перейдите по адресу http://dns name of instance:8000 и используйте комбинацию имени пользователя и пароля, которую использовали для входа в SSH.

Google Cloud Platform

В дополнение к предлагаемым экземплярам K80, P100 и V100, таким как Amazon и Azure, Google Cloud Platform (GCP) предлагает тензорные процессоры для тех, кто обладает огромными данными и высокими требованиями к вычислению. Для целей этой книги тензорные процессоры не понадобятся, однако они *будут* работать с PyTorch 1.0. Не думайте, что придется использовать TensorFlow, если проект, над которым вы работаете, требует их использования.

Начать работу с Google Cloud также довольно просто:

1. Найдите VM для глубокого обучения на GCP Marketplace.
2. Нажмите Запустить на Compute Engine.
3. Дайте экземпляру имя и присвойте его ближайшему местоположению.
4. Установите тип машины 8 vCPUs.
5. Установите графический процессор на 1 K80.
6. Убедитесь, что в разделе Framework выбран PyTorch 1.0.
7. Установите флажок Установить NVIDIA GPU автоматически при первом запуске?
8. Установите загрузочный диск на SSD-диск постоянного хранения данных.
9. Нажмите Развернуть. Для полного развертывания виртуальной машины потребуется около пяти минут.
10. Чтобы подключиться к Jupyter в экземпляре, убедитесь, что вы вошли в правильный проект в `gcloud`, и выполните эту команду:

```
gcloud compute ssh _INSTANCE_NAME_ -- -L 8080:localhost:8080
```

Плата за Google Cloud должна составлять около 70 центов в час, что делает его самым дешевым из трех основных облачных провайдеров.

Какой облачный провайдер использовать?

Если у вас нет каких-либо конкретных предпочтений, то рекомендую Google Cloud Platform (GCP); это самый бюджетный вариант, который при необходимости гораздо проще масштабировать до использования тензорных процессоров, чем предложения AWS или Azure. Но если у вас уже есть ресурсы на одной из двух других платформ, то их среды прекрасно подойдут.

Когда облачный экземпляр будет запущен, вы сможете войти в копию Jupyter Notebook. Давайте разберемся, что же делать дальше.

Использование Jupyter Notebook

Jupyter Notebook — это браузерная среда, которая позволяет комбинировать живой код с текстом, изображениями и визуализациями и которая стала одним из главных инструментов исследователей данных по всему миру. Созданными в Jupyter блокнотами легко поделиться. Скриншот Jupyter Notebook в действии показан на рис. 1.2.

В этой книге мы не будем затрагивать какие-либо расширенные функции Jupyter. Все, что нужно знать, — это как создать новый блокнот и что Shift-Enter запускает содержимое ячейки. Если вы никогда раньше не использовали его, предлагаю ознакомиться с документацией Jupyter, прежде чем переходить к главе 2.

Перед тем как приступить к работе с PyTorch, рассмотрим то, как установить все вручную.

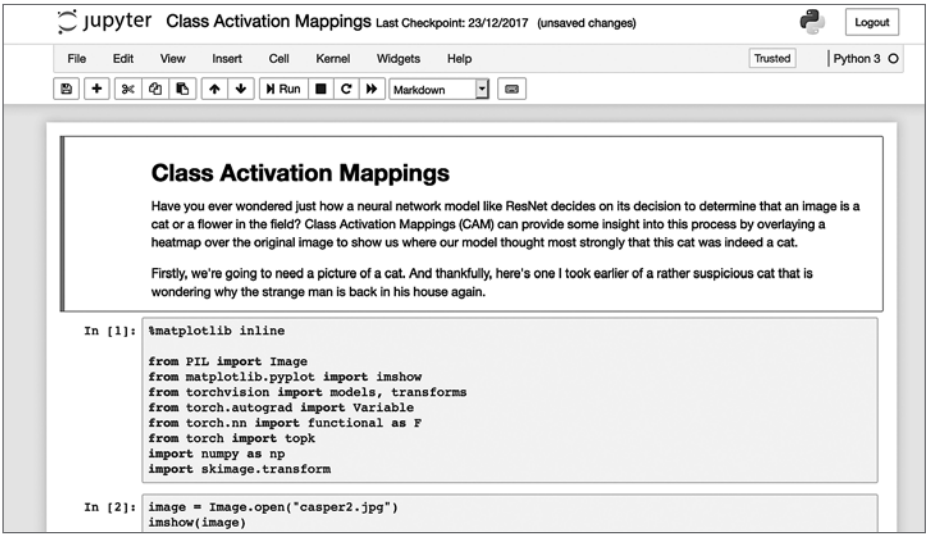

Рис. 1.2. Jupyter Notebook

Установка PyTorch с нуля

Возможно, вам требуется больше контроля своего программного обеспечения, чем просто использование одного из облачных изображений. Или коду нужна конкретная версия PyTorch. Или, несмотря на все мои предостережения, вы действительно хотите собрать собственный компьютер. Давайте узнаем, как установить PyTorch на сервере Linux.

Можно использовать PyTorch с Python 2.*x*, но настоятельно рекомендую не делать этого. Хотя эпопея обновления Python с 2.*x* до 3.*x* продолжается уже более десяти лет, все больше и больше пакетов начинают отказываться от поддержки Python 2.*x*. Поэтому, если нет веских причин этого не делать, убедитесь, что ваша система работает на Python 3.

Скачивание CUDA

Хотя PyTorch может работать в режиме CPU, в большинстве случаев для практического использования PyTorch требуется GPU, поэтому понадобится поддержка графического процессора. Это довольно просто; если есть карта NVIDIA, то настройки производятся через API-интерфейс Compute Unified Device Architecture (CUDA). Выберите на свой вкус соответствующий формат пакета Linux и установите его (https://oreil.ly/Gx_q2).

Для Red Hat Enterprise Linux (RHEL) 7:

```
sudo rpm -i cuda-repo-rhel7-10-0local-10.0.130-410.48-1.0-1.x86_64.rpm
sudo yum clean all
sudo yum install cuda
```

Для Ubuntu 18.04:

```
sudo dpkg -i cuda-repo-ubuntu1804-10-0-local-10.0.130-410.48_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-<version>/7fa2af80.pub
sudo apt-get update
sudo apt-get install cuda
```

Anaconda

У Python множество пакетов, у каждого из которых есть свои достоинства и недостатки. Как и разработчики PyTorch, я рекомендую вам установить Anaconda, обеспечивающий создание лучшего дистрибутива пакетов для специалистов по обработке данных. Как и CUDA, установить его довольно легко.

Перейдите на сайт Anaconda (https://www.anaconda.com/distribution/) и выберите установочный файл. Поскольку это огромный архив, который выполняется с помощью сценария командной оболочки, рекомендую запустить для загруженного файла `md5sum` и проверить его по списку сигнатур (https://oreil.ly/anuhu), прежде чем запускать его с помощью `bash Anaconda3-VERSION-Linux-x86_64.sh`. Это позволит убедиться, что сигнатура на вашем компьютере совпадает с сигнатурой на веб-странице. Проверка покажет, что загруженный файл не был взломан и безопасен для вашей системы. Скрипт выдаст несколько подсказок о местах установки; просто примите значения по умолчанию.

Возможно, вы задаетесь вопросом: «Можно ли сделать то же самое на MacBook?». Большинство компьютеров Mac оснащены процессорами Intel или AMD и не поддерживают PyTorch с ускорением вычислений на GPU. Советую использовать Colab или облачные технологии, а не Mac локально.

И наконец, PyTorch! (и Jupyter Notebook)

Теперь, когда у вас установлена Anaconda, настроить PyTorch проще простого:

```
conda install pytorch torchvision -c pytorch
```

Эта команда устанавливает PyTorch и библиотеку `torchvision`, которую мы будем использовать в следующих нескольких главах для создания архитектур с глубоким обучением, которые работают с изображениями. Anaconda также установила Jupyter Notebook, поэтому можно начать с ее запуска:

```
jupyter notebook
```

Перейдите по ссылке http://YOUR-IP-ADDRESS:8888, создайте новый блокнот и введите следующую команду:

```
import torch
print(torch.cuda.is_available())
print(torch.rand(2,2))
```

Должно получиться следующее:

```
True
 0.6040    0.6647
 0.9286    0.4210
[torch.FloatTensor of size 2x2]
```

Если `cuda.is_available()` возвращается в значение `False`, необходимо отладить установку CUDA, чтобы PyTorch увидел видеокарту. Значения тензора в вашем случае будут другими.

Что еще за тензор? Тензоры лежат в основе почти всего в PyTorch, поэтому нужно знать, что они собой представляют и умеют.

Тензоры

Тензор является как контейнером для чисел, так и набором правил, которые определяют преобразования между тензорами и производят новые тензоры. Проще всего представить тензоры как многомерные массивы. Каждый тензор имеет ранг, соответствующий его размерному пространству. Простой скаляр (например, 1) может быть представлен в виде тензора ранга 0, вектор — ранга 1, матрица $n \times n$ — ранга 2 и т. д. В предыдущем примере мы создали тензор ранга 2 со случайными значениями с помощью `torch.rand()`. Мы также можем создать их из списков:

```
x = torch.tensor([[0,0,1],[1,1,1],[0,0,0]])
x
>tensor([[0, 0, 1],
    [1, 1, 1],
    [0, 0, 0]])
```

Можно изменить элемент в тензоре, используя стандартное индексирование Python:

```
x[0][0] = 5
>tensor([[5, 0, 1],
    [1, 1, 1],
    [0, 0, 0]])
```

Можно использовать специальные функции создания для генерации определенных типов тензоров. В частности, `единицы()` и `нули()` будут генерировать тензоры, заполненные 1 и 0 соответственно:

```
torch.zeros(2,2)
> tensor([[0., 0.],
    [0., 0.]])
```

Можно выполнять стандартные математические операции с тензорами (например, складывать два тензора):

```
tensor.ones(1,2) + tensor.ones(1,2)
> tensor([[2., 2.]])
```

И если у вас есть тензор ранга 0, можно извлечь значение с помощью `item()`:

```
torch.rand(1).item()
> 0.34106671810150146
```

Тензоры «живут» в центральном или графическом процессоре и могут быть скопированы с помощью функции `to()`:

```
cpu_tensor = tensor.rand(2)
cpu_tensor.device
> device(type='cpu')

gpu_tensor = cpu_tensor.to("cuda")
gpu_tensor.device
> device(type='cuda', index=0)
```

Тензорные операции

В документации PyTorch говорится (https://oreil.ly/1Ev0-), что существует *множество* функций, применяемых к тензорам, — от поиска максимального элемента до применения преобразования Фурье. Чтобы превратить изображения, текст и звук в тензоры и управлять ими для выполнения наших операций, знать всего этого не нужно, но кое-какие знания определенно пригодятся. Настоятельно рекомендую ознакомиться с документацией, особенно после прочтения этой книги. Теперь мы рассмотрим все функции, которые будут использоваться в следующих главах.

Во-первых, часто нужно найти максимальный элемент в тензоре, а также *индекс*, который содержит максимальное значение (поскольку он часто соответствует классу, который нейронная сеть определила в своем окончательном прогнозе). Это делается с помощью функций `max()` и `argmax()`. Также можно использовать `item()` для получения стандартного значения Python из 1D-тензора.

```
torch.rand(2,2).max()
> tensor(0.4726)
torch.rand(2,2).max().item()
> 0.8649941086769104
```

Иногда нужно изменить тип тензора, например, `LongTensor` на `FloatTensor`. Это можно сделать через команду `to()`:

```
long_tensor = torch.tensor([[0,0,1],[1,1,1],[0,0,0]])
long_tensor.type()
> 'torch.LongTensor'
float_tensor = torch.tensor([[0,0,1],[1,1,1],[0,0,0]]).to(dtype=torch.float32)
float_tensor.type()
> 'torch.FloatTensor'
```

Большинство функций, которые работают от тензоров и возвращают тензор, создают новый тензор для хранения результата. Однако если вы хотите сэкономить память, посмотрите, определена ли функция *in-place*, имя которой должно совпадать с именем исходной функции, но с добавлением нижнего подчеркивания (_).

```
random_tensor = torch.rand(2,2)
random_tensor.log2()
>tensor([[-1.9001, -1.5013],
        [-1.8836, -0.5320]])
random_tensor.log2_()
> tensor([[-1.9001, -1.5013],
         [-1.8836, -0.5320]])
```

Еще одна распространенная операция — *изменение* тензора. Часто она осуществляется по причине того, что слою нейронной сети может потребоваться такая форма ввода, которая незначительно отличается от вводимой в настоящее время. Например, база данных образцов рукописного написания цифр, составленных Национальным институтом стандартов и технологий США (MNIST), представляет собой набор изображений 28 × 28, которые упакованы в массивах длиной 784. Чтобы использовать построенные сети, нужно преобразовать их обратно в тензоры 1 × 28 × 28 (первая цифра 1 — это количество каналов, обычно красный, зеленый и синий, но поскольку данные MNIST являются просто изображениями в оттенках серого, у нас есть только один канал). Это можно сделать через команду `view()` либо `reshape()`:

```
flat_tensor = torch.rand(784)
viewed_tensor = flat_tensor.view(1,28,28)
viewed_tensor.shape
> torch.Size([1, 28, 28])
reshaped_tensor = flat_tensor.reshape(1,28,28)
reshaped_tensor.shape
> torch.Size([1, 28, 28])
```

Обратите внимание, что форма измененного тензора должна иметь то же количество элементов, что и оригинал. Если вы введете `flat_tensor.reshape(3,28,28)`, то увидите такую ошибку:

```
RuntimeError Traceback (most recent call last)
<ipython-input-26-774c70ba5c08> in <module>()
----> 1 flat_tensor.reshape(3,28,28)

RuntimeError: shape '[3, 28, 28]' is invalid for input of size 784
```

Теперь вы можете задаться вопросом, в чем разница между `view()` и `reshape()`. Ответ заключается в том, что `view()` работает как представление исходного тензора, поэтому если базовые данные будут изменены, представление также изменится (и наоборот). Однако операция `view()` может выдавать ошибки, если требуемое представление не является *смежным*, то есть не использует тот же блок памяти, который занимало бы, если бы новый тензор требуемой формы был создан с нуля. Если это произойдет, вам нужно вызвать `tensor.contiguous()`, прежде чем вы сможете использовать операцию `view()`. Однако операция `reshape()` делает все это незаметно, поэтому в целом я рекомендую использовать `reshape()`, а не `view()`.

Наконец, может понадобиться изменить размеры тензора. Скорее всего, вы столкнетесь с этим при работе с изображениями, которые часто хранятся в виде тензоров `[высота, ширина, канал]`, но PyTorch предпочитает тензоры в виде `[канал, высота, ширина]`. В целях облегчения их обработки можно использовать команду `permute()`:

```
hwc_tensor = torch.rand(640, 480, 3)
chw_tensor = hwc_tensor.permute(2,0,1)
chw_tensor.shape
> torch.Size([3, 640, 480])
```

В данном случае мы применили операцию `permute` к тензору`[640,480,3]`, аргументы которого являются индексами размерностей тензора, поэтому нужно, чтобы конечная размерность (2, из-за нулевого индексирования) была перед нашим тензором, за которой следует две оставшиеся размерности в их обычном порядке.

Транслирование тензора

Транслирование, функция, заимствованная из NumPy, позволяет выполнять операции между одним тензором и другим, но меньшего размера. Можно транслировать через два тензора, если они начинаются в обратном порядке, то есть с конечной размерности:

- Две размерности равны.
- Одна из размерностей является 1.

В нашем случае транслирование работает, так как 1 имеет размерность 1, и поскольку другие размерности отсутствуют, 1 можно расширить, чтобы охватить другой тензор. Если бы мы попытались добавить тензор `[2,2]` в тензор `[3,3]`, то получили бы такое сообщение об ошибке:

```
Размер тензора a (2) должен соответствовать размеру тензора b (3)
при не синглетонной размерности 1
```

Но мы без проблем могли бы добавить тензор `[1,3]` в тензор `[3,3]`. Транслирование — это небольшая удобная функция, которая позволяет сделать код короче и зачастую написать его быстрее, чем прописывание вручную всех цифр и параметров тензора.

Вот и все, что нужно знать о тензорах, чтобы начать работу! Позже я расскажу о некоторых других операциях, а для чтения главы 2 имеющейся информации будет вполне достаточно.

Заключение

Итак, теперь у вас установлен PyTorch — в облаке или на локальном компьютере. Я рассказал о фундаментальном строительном блоке библиотеки — тензоре — и кратко ознакомил вас с Jupyter Notebook. Для начала всё! В следующей главе, применяя полученные знания, мы начнем создавать нейронные сети и классифицировать изображения. Прежде чем двигаться дальше, убедитесь, что тензоры и Jupyter вам понятны.

Дополнительная информация

- Документация Project Jupyter, https://jupyter.org/documentation
- Документация PyTorch, https://pytorch.org/docs/stable/
- AWS Deep Learning AMIs, https://aws.amazon.com/ru/machine-learning/amis/
- Azure Data Science Virtual Machines, https://aws.amazon.com/ru/machine-learning/amis/

ГЛАВА 2

Классификация изображений с помощью PyTorch

Учебники по глубокому обучению пестрят профессиональной непонятной терминологией. Я стараюсь свести ее к минимуму и всегда приводить один пример, который можно легко расширить, когда вы привыкнете работать с PyTorch. Мы используем этот пример на протяжении всей книги, чтобы продемонстрировать, как отладить модель (глава 7) или развернуть ее в рабочей среде (глава 8).

С этого момента и до конца главы 4 мы будем компилировать *классификатор* изображений. Нейронные сети обычно используются в качестве классификаторов изображений; сети предлагают картинку и задают простой вопрос: «Что это?».

Давайте начнем с создания нашего приложения в PyTorch.

Проблема классификации

Здесь мы создадим простой классификатор, который может отличить рыбку от кошки. Мы будем итерировать дизайн и процесс разработки нашей модели, чтобы сделать ее более точной.

На рис. 2.1 и 2.2 изображены рыбка и кошка во всей своей красе. Не уверен, есть ли у рыбки имя, а вот кошку зовут Гельветика.

Давайте начнем с обсуждения стандартных проблем, связанных с классификацией.

Рис. 2.1. Рыбка!

Рис. 2.2. Гельветика в коробке

Стандартные трудности

Как написать программу, которая сможет отличить рыбку от кошки? Возможно, вы бы написали набор правил, описывающих, что у кошки есть хвост или что у рыбки есть чешуя, и применили бы эти правила к изображению, чтобы программа классифицировала изображение. Но для этого потребуется время, усилия и навыки. А что делать, если вам попадется мэнская кошка? Хотя это явно кошка, у нее нет хвоста.

Эти правила становятся все сложнее и сложнее, когда вы пытаетесь с их помощью описать все возможные сценарии. Кроме того, должен признаться, что визуальное программирование у меня выходит кошмарно, поэтому мысль о необходимости вручную писать код для всех этих правил приводит в ужас.

Нужна функция, которая при вводе изображения возвращает кошку или рыбку. Сложно построить такую функцию, просто перечислив полностью все критерии. Но глубокое обучение, по сути, заставляет компьютер выполнять тяжелую работу по созданию всех этих правил, о которых мы только что говорили, при условии, что мы создаем структуру, предоставляем сети много данных и даем ей возможность понять, правильный ли ответ она дала. Именно это мы собираемся сделать. Кроме того, вы узнаете некоторые основные методы использования PyTorch.

Но сначала данные

Для начала нам нужны данные. Сколько данных? Зависит от разных факторов. Как вы увидите в главе 4, идея о том, что для работы любой техники глубокого обучения нужны огромные объемы данных для обучения нейронной сети, не обязательно верна. Однако сейчас мы собираемся начинать с нуля, что обычно требует доступа к большому количеству данных. Требуется много изображений рыбок и кошек.

Можно было бы потратить какое-то время на загрузку кучи изображений из поиска изображений в Google, но есть более легкий путь: стандартная коллекция изображений, используемая для обучения нейронных сетей, — *ImageNet*. Она содержит более 14 миллионов изображений и 20 тысяч категорий изображений. Это стандарт, по которому все классификаторы

изображений проводят сравнение. Поэтому я беру изображения оттуда, хотя, если хотите, то можете выбрать другие варианты.

Кроме данных у PyTorch должен быть способ определить, что такое кошка и что такое рыбка. Это достаточно просто для нас, но для компьютера задача сложнее (именно поэтому мы и создаем программу!). Мы используем *маркировку*, прикрепленную к данным, и такое обучение называется *машинное обучение с учителем*. (Если у вас нет доступа к каким-либо маркировкам, то используется, как вы наверняка догадались, *машинное обучение без учителя*.)

Если мы используем данные ImageNet, их маркировки не будут полезными, потому что содержат слишком много информации. Маркировка *полосатого кота* или *форели* для компьютера — не то же самое, что *кот* или *рыбка*.

Требуется перемаркировать их. Поскольку ImageNet представляет собой обширную коллекцию изображений, я собрал URL-адреса изображений и маркировки рыбок и кошек (https://oreil.ly/NbtEU).

Вы можете запустить скрипт *download.py* в этом каталоге, и он загрузит изображения с URL-адресов и разместит их в соответствующих местах для обучения. Перемаркировка проста; скрипт хранит изображения кошек в каталоге *train/cat* и изображения рыбок в каталоге *train/fish*. Если не хотите использовать скрипт для загрузки, просто создайте эти каталоги и разместите соответствующие изображения в нужных местах. Теперь у нас есть данные, но нужно преобразовать их в формат, понятный PyTorch.

PyTorch и загрузчики данных

Загрузка и преобразование данных в готовые к обучению форматы часто оказываются одной из областей data science, которая отнимает слишком много времени. PyTorch разработал установленные требования взаимодействия с данными, которые делают его довольно понятным, независимо от того, работаете ли вы с изображениями, текстом или аудио.

Двумя основными условиями работы с данными являются *наборы данных* и *загрузчики данных*. *Набор данных* — это класс Python, позволяющий получать данные, которые мы отправляем в нейронную сеть.

Загрузчик данных — это то, что передает данные из набора данных в сеть. (Может включать в себя такую информацию, как: *Сколько рабочих процессов передают данные в сеть*? *Сколько изображений мы передаем одновременно*?)

Давайте сначала посмотрим на набор данных. Каждый набор данных, независимо от того, содержит ли он изображения, аудио, текст, 3D-ландшафты, информацию о фондовом рынке или что-либо еще, может взаимодействовать с PyTorch, если отвечает требованиям этого абстрактного класса Python:

```
class Dataset(object):
    def __getitem__(self, index):
        raise NotImplementedError

    def __len__(self):
        raise NotImplementedError
```

Это довольно просто: мы должны воспользоваться методом, который возвращает размер нашего набора данных (`len`), и тем, который может извлечь элемент из набора данных в паре (`label, tensor`). Это вызывается загрузчиком данных, поскольку передает данные в нейронную сеть для обучения. Таким образом, мы должны написать тело для метода `getitem`, которое может взять изображение, преобразовать его в тензор и вернуть его и маркировку обратно, чтобы PyTorch мог с ним работать. Все понятно, но, очевидно, этот сценарий встречается достаточно часто, так что, может быть, PyTorch облегчит задачу?

Создание обучающего набора данных

В пакет `torchvision` включен класс `ImageFolder`, который делает практически все, при условии, что наши изображения находятся в структуре, где каждый каталог представляет собой маркировку (например, все кошки находятся в каталоге с именем *cat*). Вот что нужно для примера с кошками и рыбками:

```
import torchvision
from torchvision import transforms

train_data_path = "./train/"
```

```
transforms = transforms.Compose([
    transforms.Resize(64),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225] )
    ])

train_data = torchvision.datasets.ImageFolder
(root=train_data_path,transform=transforms)
```

Здесь добавляется кое-что еще, потому что `torchvision` также позволяет указать список преобразований, которые будут применены к изображению, прежде чем оно попадет в нейронную сеть. Преобразование по умолчанию состоит в том, чтобы взять данные изображения и превратить их в тензор (метод `trans forms.ToTensor()`, показанный в предыдущем коде), но также выполняется несколько других действий, которые могут быть не такими очевидными.

Во-первых, GPU созданы для быстрого выполнения вычислений стандартного размера. Но у нас, вероятно, есть ассортимент изображений во многих разрешениях. Чтобы повысить производительность обработки, мы масштабируем каждое входящее изображение до того же разрешения 64 × 64 с помощью преобразования `Resize` (64). Затем конвертируем изображения в тензор и, наконец, нормализуем тензор вокруг определенного набора средних и стандартных точек отклонения.

Нормализация важна, поскольку предполагается выполнение большого количества операций умножения, когда входные данные проходят через слои нейронной сети; поддержание входящих значений от 0 до 1 предотвращает серьезное увеличение значений во время фазы обучения (известное как *проблема взрывающихся градиентов*). Это волшебное воплощение — всего лишь среднее и стандартное отклонение набора данных ImageNet в целом. Можно рассчитать его специально для подмножества рыбок и кошек, но эти значения достаточно надежны. (Если бы вы работали над совершенно другим набором данных, нужно было бы вычислить это среднее значение и отклонение, хотя многие просто используют константы ImageNet и сообщают о приемлемых результатах.)

Компонуемые преобразования также позволяют легко выполнять такие действия, как поворот и сдвиг изображения для аугментации данных, к которой мы вернемся в главе 4.

В этом примере мы изменяем размеры изображений до 64 × 64. Я сделал такой случайный выбор, чтобы ускорить вычисления в нашей первой сети. Большинство существующих архитектур, которые вы увидите в главе 3, используют для своих входных изображений разрешение 224 × 224 или 299 × 299. Как правило, чем больше размер входного файла, тем больше данных, по которым сеть может обучаться. Обратная сторона медали в том, что обычно можно разместить меньшую партию изображений в памяти GPU.

О наборах данных есть множество другой информации, и это далеко не все. Но зачем нам знать больше, чем нужно, если мы уже знаем о наборе данных для обучения?

Валидация и контрольные наборы данных

Наш набор данных для обучения настроен, но теперь нужно повторить те же шаги с *набором данных для валидации*. Какая здесь разница? Одним из подводных камней глубокого обучения (и фактически всего машинного обучения) является *переобучение*: модель действительно хорошо распознает то, на чем была обучена, но не работает на примерах, которых не видела. Модель видит изображение кота, и если все другие изображения котов не очень похожи на это, модель решает, что это не кот, хотя очевидно обратное. Чтобы нейросеть так себя не вела, мы загружаем контрольную выборку в *download.py*, то есть в серию изображений кошек и рыбок, которых нет в наборе данных для обучения. В конце каждого цикла обучения (также известного как *эпоха*) мы сравниваем этот набор, чтобы убедиться, что сеть не ошиблась. Не пугайтесь, код для этой проверки невероятно прост: это тот же код с несколькими измененными именами переменных:

```
val_data_path = "./val/"
val_data = torchvision.datasets.ImageFolder(root=val_data_path,
                                            transform=transforms)
```

Мы просто использовали цепочку `transforms` вместо того, чтобы определять ее снова.

В дополнение к набору данных для валидации мы также должны создать *набор данных для проверки*. Он используется для тестирования модели после завершения всего обучения:

```
test_data_path = "./test/"
test_data = torchvision.datasets.ImageFolder(root=test_data_path,
                                             transform=transforms)
```

На первый взгляд разные типы наборов могут показаться сложными и сбивать с толку, поэтому я составил таблицу, чтобы указать, в какой части обучения используется каждый набор (табл. 2.1).

Таблица 2.1. Типы наборов данных

Набор для обучения	Используется в обучении для обновления модели
Набор для валидации	Используется для оценки того, как модель делает выводы для проблемной области, а не для обучения; не используется для непосредственного обновления модели
Набор данных для теста	Окончательный набор данных, обеспечивающий итоговую оценку производительности модели после завершения обучения

Теперь мы можем создать загрузчики данных с помощью еще нескольких строк кода на Python:

```
batch_size=64
train_data_loader = data.DataLoader(train_data, batch_size=batch_size)
val_data_loader = data.DataLoader(val_data, batch_size=batch_size)
test_data_loader = data.DataLoader(test_data, batch_size=batch_size)
```

Новое и достойное упоминания в этом коде — это команда `batch_size`. Она говорит, сколько изображений пройдет через сеть, прежде чем мы обучим и обновим ее. Теоретически мы могли бы присвоить `batch_size` ряду изображений в тестовом и обучающем наборах данных, чтобы сеть видела каждое изображение перед обновлением. На практике это обычно не делается, потому что меньшие пакеты (более широко известные в литературе как *мини-пакеты*) требуют меньше памяти и нет необходимости хранить всю информацию о каждом изображении в наборе данных, а меньший размер пакета приводит к более быстрому обучению, поскольку сеть обновляется намного быстрее. Для загрузчиков данных PyTorch значение `batch_size` по умолчанию установлено на 1. Скорее всего, вы захотите его изменить. Хотя я выбрал 64, вы можете поэкспериментировать, чтобы

понять, сколько мини-пакетов можно использовать, не исчерпав память GPU. Поэкспериментируйте с некоторыми дополнительными параметрами: например, можно указать, как будет осуществляться выборка наборов данных, будет ли перемешиваться весь набор при каждом запуске и сколько рабочих процессов задействовано для извлечения данных из набора. Все это можно найти в документации PyTorch (https://oreil.ly/XORs1).

Это касается передачи данных в PyTorch, поэтому давайте теперь представим простую нейронную сеть, которая начнет классифицировать наши изображения.

И наконец, нейронная сеть!

Мы начнем с самой простой сети глубокого обучения — входного слоя, который будет работать со входными тензорами (нашими изображениями); выходного слоя размером с число наших выходных классов (2); и скрытого слоя между ними. В первом примере будем использовать полностью связанные слои. На рис. 2.3 показан входной слой из трех узлов, скрытый слой из трех узлов и выход из двух узлов.

В этом примере каждый узел в одном слое влияет на узел в следующем слое и каждое соединение имеет вес, который определяет силу сигнала от этого узла, поступающего в следующий слой. (Именно эти веса будут обновляться, когда мы обучаем сеть, обычно из случайной инициализации.) Когда входные данные проходят через сеть, мы (или PyTorch) можем просто произвести матричное умножение весов и смещений этого слоя на входные данные. Перед передачей их в следующую функцию этот результат входит в *функцию активации*, которая является просто способом введения нелинейности в нашу систему.

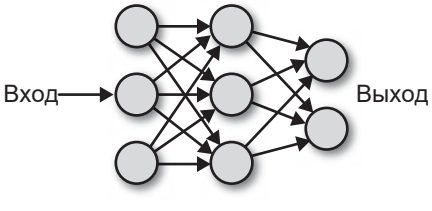

Рис. 2.3. Простая нейронная сеть

Функции активации

Функция активации — звучит мудрено, однако наиболее распространенная функция активации, которую вы сейчас можете встретить, — это ReLU, или *выпрямленный линейный блок*. Опять заумно! Но это всего лишь функция, которая реализует $max(0,x)$, поэтому результат равен 0, если входное значение отрицательное, или просто входному значению (x), если x положительное. Все просто!

Другая функция активации, с которой вы, скорее всего, столкнетесь, — это многомерная логистическая функция (softmax), которая в математическом смысле немного сложнее. По сути, она генерирует набор значений от 0 до 1, который в сумме дает 1 (вероятности!), и взвешивает значения таким образом, чтобы увеличить разность, то есть выдает один результат в векторе, который будет больше всех остальных. Вы часто будете видеть, как она используется в конце сети классификации, чтобы удостовериться, что эта сеть сделает определенный прогноз о том, к какому классу, по ее мнению, относятся входные данные.

Теперь, имея все эти строительные блоки, мы можем начать создавать нашу первую нейронную сеть.

Создание нейронной сети

Создание нейронной сети в PyTorch напоминает программирование на Python. Мы наследуем от класса под названием `torch.nn.Network` и заполняем методы `__init__` и `forward`:

```
class SimpleNet(nn.Module):

def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(12288, 84)
    self.fc2 = nn.Linear(84, 50)
    self.fc3 = nn.Linear(50,2)

def forward(self):
    x = x.view(-1, 12288)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = F.softmax(self.fc3(x))
    return x

simplenet = SimpleNet()
```

Повторюсь, это несложно. Мы делаем необходимые настройки в `init()`, в этом случае вызываем конструктор суперкласса и три полносвязных слоя (называемых в PyTorch `Linear`, в Keras они называются `Dense`). Метод `forward()` описывает, как данные передаются по сети как при обучении, так и при предсказании (*inference*). Во-первых, мы должны преобразовать трехмерный тензор (x и y плюс трехканальная цветовая информация — красный, зеленый, синий) в изображении — внимание! — в одномерный тензор, чтобы его можно было передавать в первый слой `Linear`, и мы делаем это с помощью `view()`. Таким образом мы применяем слои и функции активации по порядку, возвращая выходные данные `softmax`, чтобы получить прогноз для этого изображения.

Числа в скрытых слоях являются произвольными, за исключением выходных данных последнего слоя, который равен 2, совпадающих с нашими двумя классами — кошки или рыбки. Требуется, чтобы данные в слоях *сжимались* по мере их уменьшения в стеке. Если в слой идет, скажем, от 50 входных данных на 100 выходных данных, то сеть может *обучиться*, просто передав 50 связей на пятьдесят из ста выходных данных, и считать свою работу выполненной. Сокращая размер выходных данных по отношению ко входным данным, мы заставляем эту часть сети выучить репрезентативность исходных входных данных с меньшим количеством ресурсов, что предположительно означает, что сеть определяет некоторые отличительные признаки изображений: например, научилась распознавать плавник или хвост.

У нас есть прогноз, и можно сравнить его с фактической маркировкой исходного изображения, чтобы увидеть, был ли он правильный. Но нужен какой-то способ, позволяющий PyTorch количественно определять не только правильность или неправильность прогноза, но и то, насколько он верный или неверный. Этим занимается функция потерь.

Функции потерь

Функции потерь являются одним из ключевых элементов эффективного решения для глубокого обучения. PyTorch использует функции потерь, чтобы определить, как будет обновлять сеть для достижения желаемых результатов.

Можно сделать функцию потерь простой или сложной. PyTorch включает в себя полный набор, который охватывает большинство случаев, с которыми можно столкнуться. А еще вы всегда можете написать ее самостоятельно, если у вас нестандартный домен. Здесь мы будем использовать встроенную функцию потерь, которая называется `CrossEn tropyLoss`. Она рекомендуется для задач, связанных с категоризацией мультиклассов, подобных описываемым в этой книге. Еще одна функция потерь, с которой вы наверняка столкнетесь, — это `MSELoss`, представляющая собой стандартную среднеквадратичную ошибку, которую вы можете использовать при численном прогнозировании.

Не забывайте, что `CrossEntropyLoss` также включает в себя `soft max()`, поэтому метод `forward()` выглядит так:

```
def forward(self):
    # Преобразовать в 1D вектор
    x = x.view(-1, 12288)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

Теперь давайте посмотрим, как обновляются слои нейронной сети во время цикла обучения.

Оптимизация

Обучение сети включает в себя передачу данных через сеть, использование функции потерь для определения разности между прогнозом и фактической маркировкой, а затем использование этой информации для обновления весов сети, чтобы возврат функции потерь был как можно меньше. Для выполнения обновлений в нейронной сети мы используем *оптимизатор*.

Если бы вес был только один, мы бы построили график значения потерь относительно значения веса, который выглядел примерно так, как показано на рис. 2.4.

Если начать со случайной точки, отмеченной на рис. 2.4 знаком X, со значением веса на оси x и функцией потерь на оси y, нужно найти самую низкую точку на кривой, чтобы определить оптимальное решение. Мы

можем продвинуться вперед, изменяя значение веса, что даст новое значение функции потерь. Чтобы узнать, насколько хорошо мы продвигаемся, можно проверить градиент кривой. Один из распространенных способов изображения оптимизатора можно представить в виде перекатывания мраморного шарика, который будет стремиться к самой низкой точке (или *минимумам*) впадин. Возможно, будет понятнее, если мы создадим трехмерный график, как показано на рис. 2.5.

И в этом случае можно проверить градиенты всех возможных передвижений в каждой точке и выбрать тот, который больше всего перемещает нас вниз по гребню.

Нужно учитывать несколько важных моментов. Первое — это попадание в ловушку *локальных минимумов*, областей, которые выглядят так, как будто являются самыми низкими частями кривой потерь, если проверять градиенты, но на самом деле более низкие области находятся в других частях кривой. Если мы вернемся к нашей одномерной кривой на рис. 2.4, то увидим, что если сделать небольшой скачок вниз в минимальной точке слева, никаких причин покидать эту позицию не будет. А если мы сделаем большой скачок, то окажемся на пути, ведущем к самой низкой точке, но поскольку мы продолжаем делать такие большие скачки, то колебания сохраняются повсеместно.

Размер нашего скачка называется *скоростью обучения* и часто является *ключевым* параметром, который необходимо настроить для правильного и эффективного обучения. О способе определения хорошей скорости обучения вы узнаете в главе 4, а сейчас давайте поэкспериментируем с другими значениями: для начала попробуйте, например, 0,001. Большая скорость обучения приведет к повсеместным скачкам во время обучения и не будет сходиться с соответствующим набором весов.

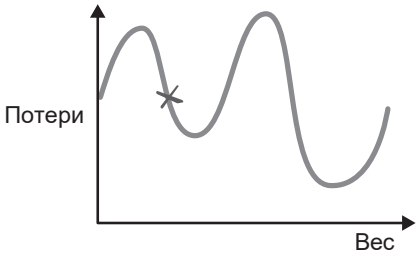

Рис. 2.4. Двумерный график потерь

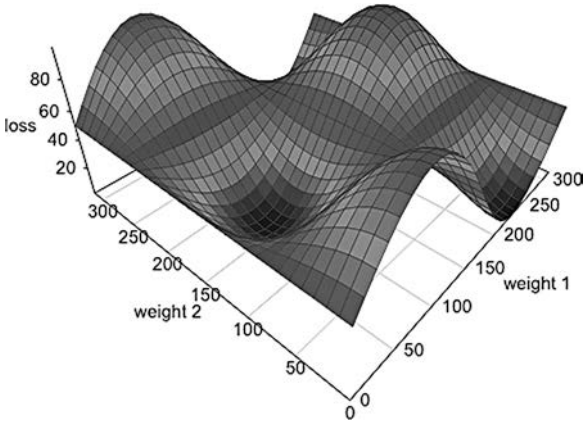

Рис. 2.5. Трехмерный график потерь

Что касается проблемы локальных минимумов, необходимо внести небольшое изменение в отбор возможных градиентов и указать выборочные случайные градиенты во время пакетирования. Это традиционный подход к оптимизации нейронных сетей и других методов машинного обучения под названием *стохастический градиентный спуск* (SGD). Но есть и другие оптимизаторы, которые лучше подходят для глубокого обучения. PyTorch включает в себя SGD и другие оптимизаторы, такие как AdaGrad и RMSProp, а также оптимизатор Adam, который мы будем преимущественно использовать.

Одна из ключевых модернизаций, которая доступна благодаря оптимизатору Adam (как и RMSProp и AdaGrad), заключается в том, что используется скорость обучения по параметру и адаптируется в зависимости от скорости изменения этих параметров.

Adam сохраняет экспоненциально убывающий список градиентов и квадрат градиентов и использует их для масштабирования глобальной скорости обучения, с которой работает. Эмпирически доказано, что Adam превосходит большинство других оптимизаторов для сетей глубокого обучения, но вы можете заменить его на SGD, RMSProp или любой другой оптимизатор, чтобы понять, дает ли использование других оптимизаторов преимущества в виде более быстрого и качественного обучения для вашего конкретного случая.

Создать оптимизатор на основе оптимизатора Adam очень просто. Мы вызываем `optim.Adam()` и передаем веса сети, которые он будет обновлять (полученные через `simplenet.parame ters()`), и наш пример скорости обучения 0,001:

```
import torch.optim as optim
optimizer = optim.Adam(simplenet.parameters(), lr=0.001)
```

Оптимизатор — это последняя часть пазла, и теперь мы наконец-то можем приступить к обучению сети.

Обучение

Вот полный цикл обучения, который включает в себя все нужное для обучения, о чем я уже писал. Мы запишем его как функцию, чтобы функцию потерь и оптимизатор можно было передавать в качестве параметров. На данный момент все выглядит довольно шаблонно:

```
for epoch in range(epochs):
    for batch in train_loader:
        optimizer.zero_grad()
        input, target = batch
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
```

Выглядит просто, но следует обратить внимание на несколько моментов. Мы берем пакет из обучающего набора на каждой итерации цикла, который обрабатывается загрузчиком данных. Затем пропускаем его через модель и вычисляем потери от ожидаемого результата. Чтобы вычислить градиенты, мы вызываем метод `backward()`. Метод `optimizer.step()` впоследствии использует эти градиенты для корректировки весов, о которых говорилось в предыдущем разделе.

Что делает `zero_grad()`? На самом деле вычисленные градиенты аккумулируются по умолчанию, а это означает, что если бы мы не обнуляли градиенты в конце итерации пакета, следующий пакет работал бы с градиентами этого пакета, а также со своими собственными градиентами, а пакету потом

пришлось бы обрабатывать два предыдущих, и т. д. Это минус, так как для оптимизации в каждой итерации нужны только градиенты текущего пакета. Мы используем `zero_grad()`, чтобы обеспечить обнуление после завершения работы с циклом.

Это упрощенная версия цикла обучения, но прежде чем мы напишем полную функцию, нужно рассмотреть кое-что еще.

Работа на GPU

Если вы уже запустили какой-либо код, то могли заметить, что он не быстрый. Как насчет замечательного GPU, установленного на нашем экземпляре в облаке (или на очень дорогом компьютере, который вы собрали)? По умолчанию PyTorch выполняет вычисления на базе CPU. Чтобы воспользоваться преимуществами GPU, нужно переместить входные тензоры и саму модель в GPU, используя метод `to()`. Вот пример, который копирует `SimpleNet` в GPU:

```
if torch.cuda.is_available():
        device = torch.device("cuda")
else
    device = torch.device("cpu")

model.to(device)
```

Теперь мы копируем модель в GPU, если PyTorch сообщает, что она доступна, или сохраняем ее в CPU иным образом. Так мы можем определить, готов ли графический процессор к работе на начальном этапе, и использовать `tensor| model.to(device)` на протяжении всей остальной программы, будучи уверенными, что код отправится в нужное место.

На этом все шаги, необходимые для обучения, завершаются. Мы почти у цели!

В более ранних версиях PyTorch использовался метод `cuda()` для копирования данных в графический процессор. Если вы сталкиваетесь с этим методом при просмотре кода, написанного другими программистами, имейте в виду, что он делает то же самое, что и `to()`!

Складываем все вместе

В этой главе вы видели много разных частей кода, поэтому давайте сложим их воедино. Мы делаем это для того, чтобы создать общий метод обучения, который охватывает модель, а также обучение и набор данных для валидации, включая параметры скорости обучения и размера партии. Мы будем использовать следующий код в остальной части книги:

```
def train(model, optimizer, loss_fn, train_loader, val_loader,
epochs=20, device="cpu"):
    for epoch in range(epochs):
        training_loss = 0.0
        valid_loss = 0.0
        model.train()
    for batch in train_loader:
        optimizer.zero_grad()
        inputs, target = batch
        inputs = inputs.to(device)
        target = targets.to(device)
        output = model(inputs)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
        training_loss += loss.data.item()
training_loss /= len(train_iterator)

model.eval()
num_correct = 0
num_examples = 0
for batch in val_loader:
    inputs, targets = batch
    inputs = inputs.to(device)
    output = model(inputs)
    targets = targets.to(device)
    loss = loss_fn(output,targets)
    valid_loss += loss.data.item()
    correct = torch.eq(torch.max(F.softmax(output), dim=1)[1],
                                                    target).view(-1)
    num_correct += torch.sum(correct).item()
    num_examples += correct.shape[0]
valid_loss /= len(valid_iterator)

print('Epoch: {}, Training Loss: {:.2f},
Validation Loss: {:.2f},
accuracy = {:.2f}'.format(epoch, training_loss,
valid_loss, num_correct / num_examples))
```

Это наша функция обучения, и можно приступать к обучению, вызвав его с необходимыми параметрами:

```
train(simplenet, optimizer, torch.nn.CrossEntropyLoss(),
      train_data_loader, test_data_loader,device)
```

Обучение сети займет 20 эпох (вы можете настроить эти параметры, передав значение для `epoch` в `train()`), и в конце каждой эпохи нужно получить результаты точности модели для контрольной выборки.

Вы обучили свою первую нейронную сеть — мои поздравления! Теперь ее можно использовать для прогнозирования, поэтому давайте разберемся, как это делается.

Прогнозирование

Еще в начале главы я сказал, что мы создаем нейронную сеть, которая могла бы определить, изображена на картинке кошка или рыбка. Сейчас мы обучили нейронную сеть, но что делать дальше, чтобы сгенерировать прогноз для одного изображения?

Вот небольшой фрагмент кода на языке Python, который загрузит изображение из файловой системы и покажет, говорит наша сеть «*кошка*» или «*рыбка*»:

```
from PIL import Image

labels = ['cat','fish']

img = Image.open(FILENAME)
img = transforms(img)
img = img.unsqueeze(0)

prediction = simplenet(img)
prediction = prediction.argmax()
print(labels[prediction])
```

Бóльшая часть этого кода проста; мы повторно используем конвейер преобразования, созданный нами ранее, чтобы трансформировать изображение в правильную форму для нейронной сети. Но поскольку сеть использует пакеты, то фактически ожидает четырехмерный тензор, где первая размерность обозначает различные изображения в пакете. У нас

нет пакета, но можно создать пакет длиной 1, используя `unsqueeze(0)`, добавляющий новую размерность в начало нашего тензора.

Получение прогнозов так же просто, как передача *партии* в модель. Затем мы должны определить класс с большей вероятностью. В этом случае можно просто конвертировать тензор в массив и сравнить два элемента, но зачастую их гораздо больше. К счастью, у PyTorch есть функция `argmax()`, которая возвращает индекс наибольшего значения тензора. Затем мы используем ее для индексации в нашем массиве маркировок и получения прогноза. В качестве тренировки используйте предыдущий код как основу разработки прогнозов для набора тестовых данных, о котором я писал в начале главы. Вам не нужно использовать `unsqueeze()`, потому что вы получаете пакеты из `test_data_loader`.

Это все, что вам нужно знать о прогнозировании на данном этапе; мы вернемся к нему в главе 8, когда информация для разработки станет сложнее.

В дополнение к прогнозированию вы, скорее всего, захотите иметь возможность перезагружать модель в любое время, сохранив обученные параметры, поэтому давайте разберемся, как это делается в PyTorch.

Сохранение модели

Если производительность модели вас устраивает или по какой-либо причине вам необходимо приостановить работу, можно сохранить текущее состояние модели в формате консервации (*pickle*) Python с помощью метода `torch.save()`. И наоборот, можно загрузить ранее сохраненную итерацию модели, используя метод `torch.load()`.

Сохранение текущих параметров и структуры модели будет выглядеть так:

```
torch.save(simplenet, "/tmp/simplenet")
```

Перезагрузить же можно следующим образом:

```
simplenet = torch.load("/tmp/simplenet")
```

Так вы сохраняете и параметры, и структуру модели в файл. Если вы измените структуру модели на более позднем этапе, могут возникнуть сложности. Поэтому более распространенной практикой сохранения обу-

ченной модели является `state_dict`. Это стандартный словарь `dict` Python, который содержит карты параметров каждого слоя в модели. Сохранение `state_dict` выглядит так:

```
torch.save(model.state_dict(), PATH)
```

Чтобы восстановить модель, сначала создайте экземпляр модели, а затем используйте `load_state_dict`. Для `SimpleNet`:

```
simplenet = SimpleNet()
simplenet_state_dict = torch.load("/tmp/simplenet")
simplenet.load_state_dict(simplenet_state_dict)
```

Преимущество в том, что если вы каким-то образом расширяете модель, то можете задать параметр `strict=False` для `load_state_dict`, который присваивает параметры слоям в модели, существующим в `state_dict`, но не выдает ошибку, если в загруженном `state_dict` отсутствуют слои или они добавлены из текущей структуры модели. Поскольку это просто обычный словарь Python `dict`, можно изменить ключевые имена в соответствии с моделью, что будет удобно в случае вытаскивания параметров из совершенно другой модели.

Модели можно сохранять на диск во время обучающего цикла и перезапускать в другой момент, таким образом, обучение может продолжиться с того места, где закончилось. Это весьма полезная опция при использовании чего-то вроде Google Colab, который дает постоянный доступ к GPU в течение приблизительно 12 часов. Отслеживая время, вы можете сохранить модель до отключения и продолжить обучение в новой 12-часовой сессии.

Заключение

Мы кратко познакомились с основами нейронных сетей и узнали, как можно обучать с помощью PyTorch, используя набор данных, делать прогнозы для других изображений и сохранять/восстанавливать модели на диск и с диска.

Прежде чем перейти к следующей главе, поэкспериментируйте с архитектурой `SimpleNet`, которую мы здесь создали. Отрегулируйте количество

параметров в слоях `Linear` и по возможности добавьте еще один или два слоя. Взгляните на различные функции активации, доступные в PyTorch, и поменяйте `ReLU` на что-то другое. Посмотрите, что произойдет, если вы отрегулируете скорость обучения или используете другой оптимизатор вместо Adam (попробуйте базовый SGD). Возможно, нужно изменить размер пакета и исходный размер изображения, поскольку в начале прямого прохода он преобразуется в одномерный тензор.

Глубокое обучение все еще находится в развивающейся стадии; отладка скорости обучения выполняется вручную до тех пор, пока сеть не будет обучена надлежащим образом, поэтому хорошо бы понять, как взаимодействуют все ее части.

Вас может разочаровать точность архитектуры `SimpleNet`, но не расстраивайтесь! В главе 3 вы узнаете о некоторых хитростях и увидите сверточную нейронную сеть вместо очень простой сети, которую мы использовали до сих пор.

Дополнительные источники

- Документация PyTorch, https://pytorch.org/docs/stable/index.html
- «Adam: A Method for Stochastic Optimization», Diederik P. Kingma and Jimmy Ba (2014), https://arxiv.org/abs/1412.6980
- «An Overview of Gradient Descent Optimization Algorithms», Sebastian Ruder (2016), https://arxiv.org/abs/1609.04747

ГЛАВА 3

Сверточные нейронные сети

Поэкспериментировав с полносвязными нейронными сетями в главе 2, вы наверняка кое-что заметили. Если попытаться добавить больше слоев или значительно увеличить количество параметров, GPU, скорее всего, не хватит памяти. Кроме того, понадобилось некоторое время, чтобы обучить сеть чему-то такому, что дало бы приличную точность, и даже этого бывает недостаточно, особенно если учесть все эти восторги по поводу глубокого обучения. Так в чем же дело?

Полносвязная (упреждающая) сеть действительно может функционировать в качестве универсального аппроксиматора, но теория не говорит о том, сколько времени понадобится, чтобы обучить сеть такой аппроксимации к функции, которая действительно нужна. Но мы можем все улучшить, особенно если речь идет об изображениях. В этой главе вы узнаете о *сверточных нейронных сетях* (CNN) и о том, как они образуют основу самых точных классификаторов изображений на сегодняшний день (и рассмотрим некоторые из них). Мы создадим новую архитектуру на основе сверточных сетей для нашего приложения «рыбка или кошка», и вы увидите, что сеть может обучаться быстрее *и* точнее, чем в предыдущей главе. Давайте начнем!

Первая сверточная модель

На этот раз я познакомлю вас с архитектурой окончательной модели, а затем дам новую информацию. И как я упоминал в главе 2, метод обучения, который мы создали, не зависит от модели, поэтому можете сначала протестировать эту модель, а затем прочитать пояснения!

```
class CNNNet(nn.Module):

    def __init__(self, num_classes=2):
```

```
        super(CNNNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(),
            nn.Linear(4096, num_classes)
        )
    def forward(self, x):
      x = self.features(x)
      x = self.avgpool(x)
      x = torch.flatten(x, 1)
      x = self.classifier(x)
      return x
```

Первое, на что нужно обратить внимание, — использование модуля `nn.Sequential()`. Так мы можем создавать цепочку слоев. Когда мы используем одну из этих цепочек в `forward()`, входные данные последовательно проходят через каждый элемент массива слоев. Таким образом вы можете разбить вашу модель на более логичные упорядоченные структуры данных. В этой сети у нас есть две цепочки: блок `features` и блок `classifier`. Давайте посмотрим на новые слои, которые мы вводим, начиная с `Conv2d`.

Свертки

Слой `Conv2d` — это двумерная *свертка*. Если у нас есть изображение в оттенках серого, оно состоит из массива шириной x пикселов и высотой y пикселов, причем каждый вход имеет значение, указывающее на то,

является он черным или белым или чем-то средним (мы берем за основу 8-битное изображение, поэтому каждое значение может варьироваться от 0 до 255). В этом примере мы рассмотрим небольшое квадратное изображение высотой и шириной 4 пиксела:

$$\begin{bmatrix} 10 & 11 & 9 & 3 \\ 2 & 123 & 4 & 0 \\ 45 & 237 & 23 & 99 \\ 20 & 67 & 20 & 255 \end{bmatrix}$$

Далее мы вводим нечто под названием *фильтр*, или *ядро свертки*. Это еще одна матрица, скорее всего, меньшего размера, которую мы будем перетаскивать по нашему изображению. Вот наш фильтр 2 × 2:

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

Для получения выходных данных мы берем меньший фильтр и накладываем его поверх исходных входных данных, как увеличительное стекло над листом бумаги. Начиная с верхнего левого угла, первый расчет выглядит следующим образом:

$$\begin{bmatrix} 10 & 11 \\ 2 & 123 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

И все, что мы делаем, — это умножаем каждый элемент в матрице на его соответствующий член в другой матрице и складываем результат: `(10 × 1)` + + `(11 × 0)` + `(2 × 1)` + `(123 × 0)` = `12`. Затем перемещаем фильтр и начинаем снова. Но насколько нужно переместить фильтр? В этом случае мы перемещаем фильтр на 2, таким образом, наш второй расчет выглядит так:

$$\begin{bmatrix} 9 & 3 \\ 4 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

В результате мы получаем значение 13. Теперь мы переместим фильтр вниз и назад влево и повторим процесс, получив окончательный результат (или *карту признаков*):

$$\begin{bmatrix} 12 & 13 \\ 65 & 45 \end{bmatrix}$$

На рис. 3.1 вы можете увидеть, как это работает: ядро 3 × 3 перетаскивается через тензор 4 × 4 и создает выходные данные 2 × 2 (хотя в основе каждого сегмента девять элементов, а не четыре, как в первом примере).

В слое свертки будет много таких фильтров, значения которых заполняются по мере обучения сети, и все фильтры в слое имеют одинаковые значения смещения. Давайте вернемся к тому, как мы вызываем слой `Conv2d`, и рассмотрим некоторые другие параметры, которые можно задать:

```
nn.Conv2d(in_channels,out_channels, kernel_size, stride, padding)
```

`in_channels` — это количество входных каналов, которые мы будем получать в слое. В начале сети мы принимаем RGB-изображение в качестве входных данных, поэтому количество входных каналов равно трем. `out_channels` — это число выходных каналов, которое соответствует количеству фильтров в нашем слое. Далее следует `kernel_size`, который описывает высоту и ширину нашего фильтра.[1] Это может быть отдельный скаляр, задающий квадрат (например, в первом слое мы задаем фильтр 11 × 11), или вы можете использовать кортеж (например, (3,5) для фильтра 3 × 5). Следующие два параметра кажутся достаточно простыми, но они могут значительно повлиять на нижестоящие уровни вашей сети и даже на то, куда в конечном итоге будет обращен этот конкретный уровень. `stride` показывает, на сколько шагов по всем входным данным мы продвигаемся, когда настраиваем фильтр на новую позицию. В нашем примере мы получаем шаг 2, что приводит к эффекту создания карты признаков, равной половине входных данных.

Но мы могли бы также продвигаться с шагом 1, что дало бы нам карту признаков выхода 4 × 4, тот же размер ввода. Мы также можем передать кортеж (*a, b*), что позволило бы нам перемещать *a* поперек и *b* вниз на каждом шаге. Теперь вам должно быть интересно, что произойдет, когда он дойдет до конца? Давайте посмотрим. Если мы перетащим наш фильтр с шагом 1, то в конечном итоге получим следующее:

$$\begin{bmatrix} 3 & ? \\ 0 & ? \end{bmatrix}$$

[1] В литературе термины «ядро» и «фильтр», как правило, взаимозаменяемы. Если у вас есть опыт работы с графикой, то вы, скорее всего, знаете, что такое ядро. Лично я предпочитаю термин «фильтр».

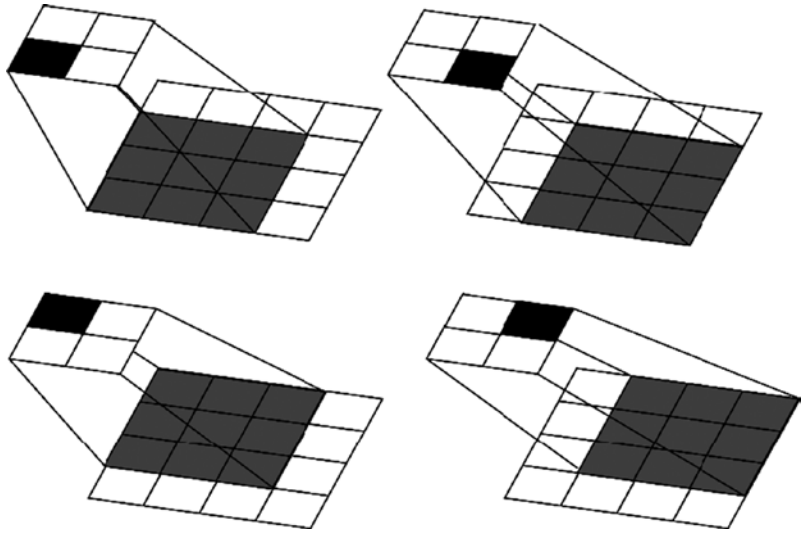

Рис. 3.1. Как ядро 3 × 3 работает со входом 4 × 4

В нашем входе недостаточно элементов для полной свертки. Так что же происходит? Вот здесь и вступает в игру параметр `padding (отступ)`. Если значение `padding` будет 1, то ввод будет выглядеть примерно так:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & 11 & 9 & 3 & 0 \\ 0 & 2 & 123 & 4 & 0 & 0 \\ 0 & 45 & 237 & 23 & 99 & 0 \\ 0 & 20 & 67 & 22 & 255 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Теперь, когда мы добрались до конца, значения, охватываемые фильтром, выглядят следующим образом:

$$\begin{bmatrix} 3 & 0 \\ 0 & 0 \end{bmatrix}$$

Если вы не зададите отступ, любые пограничные случаи, с которыми столкнется PyTorch в последних столбцах ввода, просто отбрасы-

ваются. Вы сами решаете, каким должен быть отступ. Как и в случае с `stride` и `kernel_size`, вы также можете передать кортеж для отступа `height` × `weight`, а не отдельное число, которое заполняется одинаково в обоих направлениях.

Вот что делают слои `Conv2d` в нашей модели. А что насчет слоев `Max Pool2d`?

Субдискретизация

Вы часто будете видеть слои пулинга (субдискретизации) в сочетании со слоями свертки. Эти слои снижают разрешение сети от предыдущего входного слоя, что дает нам меньше параметров на нижних слоях. Такое сжатие приводит к более быстрым вычислениям вначале и помогает предотвратить переобучение сети.

В нашей модели мы используем `MaxPool2d` с размером ядра 3 и шагом 2. Давайте посмотрим, как это работает, на примере. Вход 5 × 3:

$$\begin{bmatrix} 1 & 2 & 1 & 4 & 1 \\ 5 & 6 & 1 & 2 & 5 \\ 5 & 0 & 0 & 9 & 6 \end{bmatrix}$$

Используя размер ядра 3 × 3 и шаг 2, мы получаем два тензора 3 × 3 от субдискретизации:

$$\begin{bmatrix} 1 & 2 & 1 \\ 5 & 6 & 1 \\ 5 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 4 & 1 \\ 1 & 2 & 5 \\ 0 & 9 & 6 \end{bmatrix}$$

В `MaxPool` мы берем максимальное значение каждого из этих тензоров, что дает выходной тензор [6, 9]. Как и в слоях свертки, в `Max Pool` есть опция дополнительных гиперпараметров для управления степенью дополнения

входного тензора данных нулями (`padding`), которая создает границу нулевых значений вокруг тензора в случае, если шаг выходит за пределы окна тензора.

Таким образом, можно объединить другие функции, а не только получать максимальное значение из ядра. Не менее распространенным методом является получение среднего значения тензора, которое позволяет всем тензорным данным вносить вклад в субдискретизацию, а не задействовать только одно значение в случае с `max` (и если вы подумаете об изображении, то можете предположить, что понадобится учитывать ближайших соседей пиксела). Кроме того, в PyTorch есть слои `AdaptiveMaxPool` и `AdaptiveAvgPool`, которые работают независимо от размеров входящего тензора (например, в нашей модели есть `AdaptiveAvgPool`). Советую вам использовать их в модельных архитектурах, которые вы создаете поверх стандартных слоев `MaxPool` или `AvgPool`, поскольку они позволяют создавать архитектуры, которые могут работать с различными входными измерениями; это удобно при работе с несопоставимыми наборами данных.

У нас есть еще один новый компонент, который невероятно прост, но важен для обучения.

Прореживание, или дропаут

Одной из проблем нейронных сетей является их склонность к переобучению, поэтому в мире глубокого обучения ведется серьезная работа над выявлением подходов, которые позволят сетям учиться и обобщать данные, не связанные с обучением, без обучения тому, как реагировать на входные данные для тренировки. Слой `Dropout` — чертовски простой способ решить эту задачу, кроме того, его преимущество заключается в простоте его понимания и эффективности: что, если мы просто не будем обучать случайный набор узлов сети во время цикла обучения? Поскольку они не будут обновляться, у них не будет возможности соответствовать исходным данным, и поскольку они случайные, каждый цикл обучения будет игнорировать различный выбор входных данных, что должно способствовать обобщению еще больше.

В нашем примере сети CNN слои `Dropout` по умолчанию инициализируются с 0,5, а это значит, что 50 % входного тензора обнуляется случайным об-

разом. Если вы хотите изменить это значение на 20 %, добавьте параметр `p` к вызову инициализации: `Dropout(p=0.2)`.

`Dropout` должен происходить только во время обучения. Если бы он происходил во время вывода, сеть бы поглупела, а это не то, к чему мы стремимся! К счастью, реализация `Dropout` в PyTorch определяет, в каком режиме вы работаете, и передает все данные через слой `Dropout` во время вывода.

Изучив нашу маленькую модель CNN и типы слоев, давайте посмотрим на другие модели, созданные за последние десять лет.

История архитектур CNN

Хотя модели CNN существуют уже несколько десятилетий (например, LeNet-5 использовался для распознавания цифр при проверке в конце 1990-х годов), только со временем, когда GPU стали широкодоступны, глубокие сети CNN стали использоваться повсеместно. Это произошло через семь лет после того, как сети глубокого обучения начали преобладать над всеми другими существующими подходами в классификации изображений. В этом разделе мы отправимся в небольшое путешествие во времени, назад в прошлое, чтобы поговорить о вехах обучения на основе CNN и изучить некоторые новые методики.

AlexNet

Во многих отношениях *AlexNet* была архитектурой, которая изменила все. Она была выпущена в 2012 году и уничтожила все остальные данные ImageNet в конкурсе по распознаванию объектов того года: эта сеть уменьшила частоту непопадания в первые пять ошибок (top-5 error rate) до 15,3 % (второе место заняла архитектура с ошибкой всего 26,2 %, просто чтобы вы представляли, насколько лучше остальных она была). AlexNet был одной из первых архитектур, которая представила концепции `MaxPool` и `Dropout` и даже популяризировала тогда еще менее известную функцию

активации `ReLU`. Это была одна из первых архитектур, продемонстрировавших, что использовать множественные слои для обучения графического процессора было не только возможно, но и эффективно. Хотя это уже и не передовые технологии, AlexNet остается важной вехой в истории глубокого обучения.

Как выглядит архитектура AlexNet? Вот и пришло время раскрыть маленький секрет. Какую сеть мы использовали в этой главе? Ответ: AlexNet. Сюрприз! Чтобы соответствовать первоначальному определению AlexNet, мы использовали стандартный `MaxPool2d` вместо `AdaptiveMaxPool2d`.

Inception/GoogLeNet

Давайте перейдем к победителю конкурса ImageNet 2014 года. Архитектура GoogLeNet представила модуль *Inception*, который устраняет некоторые недостатки AlexNet. В этой сети ядра слоев свертки фиксируются с определенным разрешением. Мы можем ожидать, что изображение будет иметь детали, которые важны как в макро-, так в микромасштабе. При большем ядре, возможно, будет легче определить, является ли объект автомобилем, но чтобы определить, является ли он внедорожником или хэтчбеком, может потребоваться меньшее ядро. Для определения модели нам может понадобиться ядро еще меньшего размера, чтобы выявить такие детали, как логотипы и знаки различия.

Вместо этого сеть Inception запускает серию сверток разных размеров на одном входе и соединяет все фильтры вместе, чтобы перейти к следующему слою.

Однако прежде чем выполнить какое-либо из этих действий, она выполняет свертку 1×1 в качестве узкого места (*bottleneck*), сжимающего входной тензор, а это значит, что ядра 3×3 и 5×5 работают на меньшем количестве фильтров, чем если бы не было свертки 1×1. Вы можете увидеть модуль Inception на рис. 3.2.

Исходная архитектура GoogLeNet использует девять из этих модулей, расположенных друг над другом, образуя глубокую сеть. Несмотря на всю глубину, она в целом использует меньше параметров, чем AlexNet, обеспечивая при этом производительность с ошибкой 6,67 %, что сравнимо с человеком.

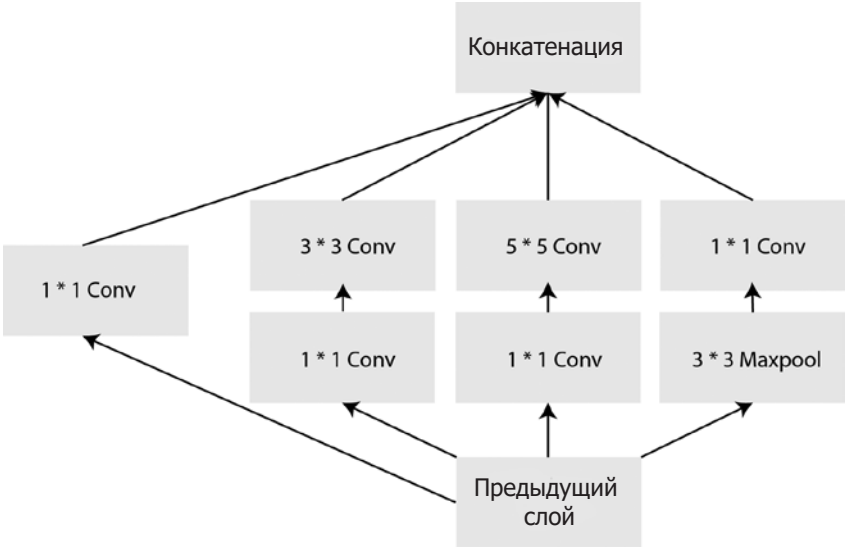

Рис. 3.2. Модуль Inception

VGG

Второе место в соревновании ImageNet 2014 года занял Оксфордский университет и его сеть Visual Geometry Group (VGG). В отличие от GoogLeNet, VGG представляет собой более простой стек слоев свертки. Предлагая различные конфигурации более длинных стеков сверточных фильтров в сочетании с двумя большими скрытыми линейными слоями перед финальным классификационным слоем, он демонстрирует силу простых глубоких архитектур (выдавая 8,8 % ошибок в своей конфигурации VGG-16). На рис. 3.3 показаны слои VGG-16 от начала до конца.

Недостатком подхода VGG является то, что конечные полносвязные слои раздувают сеть до большого размера весом в 138 миллионов параметров по сравнению с 7 миллионами GoogLeNet. При этом, несмотря на свой огромный размер, который легко представить из-за ее более простой конструкции и ранней доступности обученных весов, в мире глубокого обучения сеть VGG все еще остается довольно популярной. VGG часто используется для переноса стиля (например, превращение фотографии в картину Ван Гога), так как благодаря комбинации ее сверточных фильтров это делается проще, чем если бы мы использовали более сложные сети.

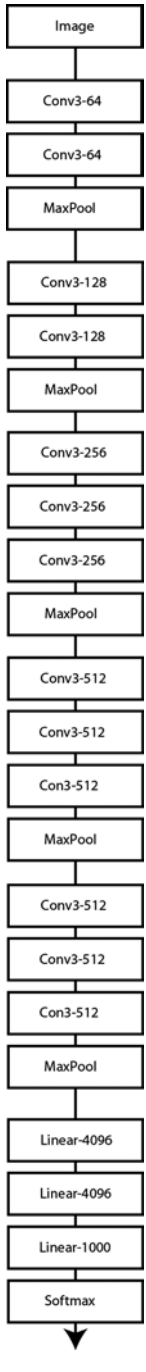
Image
Conv3-64
Conv3-64
MaxPool
Conv3-128
Conv3-128
MaxPool
Conv3-256
Conv3-256
Conv3-256
MaxPool
Conv3-512
Conv3-512
Con3-512
MaxPool
Conv3-512
Conv3-512
Con3-512
MaxPool
Linear-4096
Linear-4096
Linear-1000
Softmax

Рис. 3.3. VGG-16

ResNet

Год спустя архитектура ResNet от Microsoft выиграла конкурс ImageNet 2015, достигнув 4,49 % из топа-5 ошибок валидации в модели ResNet-152 и 3,57 % в ансамблевой модели (что на данный момент значительно превосходит возможности человека). Новшество, которое привнесла архитектура ResNet, представляло собой усовершенствованный по сравнению с модулем Inception подход к пакетированию слоев, в котором каждый пакет выполнял обычные операции CNN, а также добавлял входящий ввод к выходу блока, как показано на рис. 3.4.

Преимущество состоит в том, что каждый блок проходит через исходный вход на следующий слой, что позволяет «сигналу» обучающих данных проходить через более глубокие сети, чем это возможно в VGG или Inception. Эта потеря изменения веса в глубоких сетях известна как «*исчезающий*» *градиент,* так как градиент изменяется в обратном распространении, стремясь к нулю во время процесса обучения.

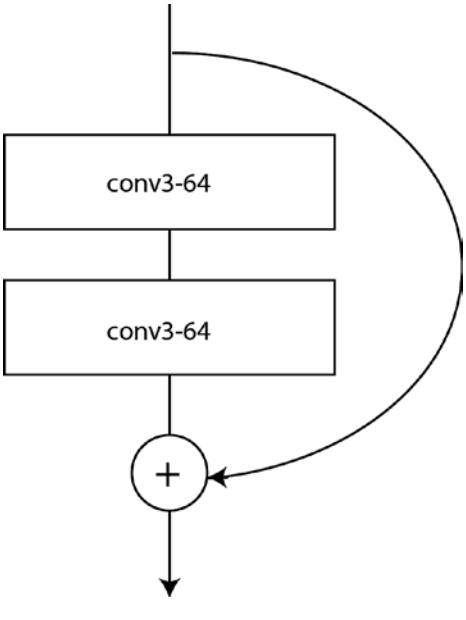

Рис. 3.4. Блок ResNet

Другие архитектуры тоже доступны!

Примерно с 2015 года множество других архитектур постепенно показывали более высокую точность на ImageNet, например DenseNet (расширенная идея ResNet, позволяющая создавать 1000-слойных монстров), кроме того, было проделано много работы в создании таких архитектур, как SqueezeNet и MobileNet, которые предлагают достаточную точность, но по сравнению с VGG, ResNet или Inception являются крошечными.

Еще одна важная область исследований — это поиск способа заставить нейронные сети самостоятельно начать разрабатывать нейронные сети. На данный момент самая успешная попытка, конечно, предпринята Google, чья система AutoML создала архитектуру под названием *NASNet* с ошибкой топ-5 размером 3,8 % в ImageNet. На момент написания этой книги, а именно начало 2019 года, она является самой современной технологией (наряду с другой автоматически сгенерированной архитектурой от Google под названием *PNAS*). Фактически организаторы конкурса ImageNet решили прекратить дальнейшие соревнования в этой сфере, потому что архитектуры уже вышли за пределы уровня человеческих способностей.

Таким образом, мы с вами подошли к самым современным технологиям на момент выхода этой книги в печать, поэтому давайте посмотрим, как мы можем использовать эти модели, вместо того чтобы определять свои собственные.

Использование предварительно обученных моделей в PyTorch

Очевидно, что определять модель каждый раз, когда вы хотите ее использовать, было бы весьма рутинно, особенно если вы выходите из AlexNet, поэтому PyTorch по умолчанию предоставляет многие из самых популярных моделей в библиотеке `torchvision`. В AlexNet все, что вам нужно сделать, — это следующее:

```
import torchvision.models as models
alexnet = models.alexnet(num_classes=2)
```

Определения для вариантов VGG, ResNet, Inception, DenseNet и SqueezeNet также доступны. Это дает вам определение модели, но вы также можете пойти дальше и вызвать `models.alexnet(pretrained=True)`, чтобы загрузить предварительно обученный набор весов для AlexNet, что позволит вам сразу использовать его для классификации без дополнительного обучения. (Но, как вы увидите в следующей главе, скорее всего, вы захотите выполнить дополнительное обучение, чтобы повысить точность конкретного набора данных.)

Кроме того, есть кое-что еще, о чем следует упомянуть, говоря о самостоятельном построении моделей, чтобы вы прочувствовали, как они сочетаются друг с другом. Это хороший способ получить некоторые практические навыки построения архитектурной модели в PyTorch, и конечно, вы можете сравнить их с предоставленными моделями, чтобы убедиться, что ваши задумки соответствуют фактическому определению. Но как узнать, что это за структура?

Изучение структуры модели

Если вам интересно, как устроена одна из этих моделей, есть простой способ заставить PyTorch помочь вам. В качестве примера рассмотрим целую архитектуру ResNet-18, которую мы получаем, просто вызывая следующее:

```
print(model)

ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
  bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1,
  dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
       track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
       track_running_stats=True)
```

```
  )
  (1): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
     padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
    (relu): ReLU(inplace)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
    padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
  )
)
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
     padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
    (relu): ReLU(inplace)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
     padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
       bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
       track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
     padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
    (relu): ReLU(inplace)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
     padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
     track_running_stats=True)
  )
)
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
     padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
    track_running_stats=True)
    (relu): ReLU(inplace)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
    padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
```

```
      track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
        bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
      padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
        bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
        track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
       track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
      padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
      track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

В этом коде нет практически ничего из того, чего вы еще не встречали в этой главе, за исключением `BatchNorm2d`. Давайте посмотрим, что он делает в одном из этих слоев.

Пакетная нормализация (BatchNorm)

BatchNorm, сокращение от «пакетная нормализация», — это простой слой с единственной задачей: использование двух обученных параметров (то есть он будет обучаться вместе с остальной частью сети), чтобы убедиться, что каждый мини-пакет, проходящий через сеть, имеет нулевое математическое ожидание и единичную дисперсию. Зачем это нужно, если мы уже нормализовали входные данные, используя цепочку преобразования (см. главу 2)?

Для небольших сетей `BatchNorm` действительно менее полезен, но по мере их увеличения влияние любого слоя на другой слой, скажем, на 20 слоев ниже, может быть значительным из-за многократного умножения, и вы можете получить либо исчезающие, либо взрывающиеся градиенты, являются катастрофой для процесса обучения. Слои `BatchNorm` гарантируют, что даже если вы используете такую модель, как ResNet-152, умножения внутри вашей сети не выйдут из-под контроля.

Возникает вопрос: если в нашей сети есть `BatchNorm`, то почему мы вообще нормализуем входные данные в цепочке преобразования цикла обучения? В конце концов, разве `BatchNorm` не должен работать за нас? Отвечаю: да, вы могли бы так сделать! Но сети потребуется больше времени, чтобы научиться контролировать входные данные, поскольку им придется самостоятельно обнаруживать первоначальное преобразование, что увеличивает время обучения.

Советую инстанцировать все архитектуры, о которых мы говорили ранее, и использовать `print(model)`, чтобы увидеть, какие слои они используют и в каком порядке выполняются операции. После этого возникает еще один ключевой вопрос: *какую из этих архитектур следует использовать*?

Какую модель мне использовать?

Мой непрактичный ответ таков: естественно, ту, которая вам подходит! Но давайте копнем немного глубже. Во-первых, хотя я и предлагаю вам попробовать архитектуры NASNet и PNAS, я не буду вам их советовать, несмотря на их впечатляющие результаты в ImageNet. При работе они могут требовать на удивление много памяти, а методика *переноса обучения*, о которой вы узнаете в главе 4, не столь эффективна по сравнению с созданными человеком архитектурами, включая ResNet.

Предлагаю вам взглянуть на Kaggle — веб-сайт, который проводит сотни конкурсов по исследованию данных, и посмотреть, что используют победители. Скорее всего, вы увидите множество ансамблей на основе ResNet. Лично я предпочитаю архитектуры ResNet всем остальным перечисленным здесь, во-первых, потому что они предлагают хорошую точность, а во-вторых, потому что с ними легко начать экспериментировать с моделью ResNet-34 для быстрой итерации, а затем перейти к большим сетям ResNet (и, что более реалистично, к ансамблю различных архитектур ResNet, таких, которые использовала компания Microsoft на конкурсе ImageNet в 2015 году), если я чувствую, что у меня есть что-то перспективное.

Прежде чем мы закончим главу, поделюсь последними новостями, касающимися скачивания предварительно обученных моделей.

Необходимые покупки: PyTorch Hub

Недавняя новинка из мира PyTorch предлагает дополнительный способ получения моделей: *PyTorch Hub*. Предполагается, что в будущем он станет центральным источником получения любой опубликованной модели, будь то работа с изображениями, текстом, аудио, видео или данными любого другого типа. Чтобы получить модель, используйте модуль `torch.hub`:

```
model = torch.hub.load('pytorch/vision', 'resnet50', pretrained=True)
```

Первый параметр указывает на владельца GitHub и репозиторий (с опциональным идентификатором тега/ветви в строке); второй — запрашиваемая модель (в данном случае `resnet50`); и, наконец, третий указывает, следует ли загружать предварительно обученные веса. Также можно использовать `torch.hub.list('pytorch/vision')`, чтобы найти все доступные для загрузки модели в этом репозитории.

На середину 2019 года PyTorch Hub является новинкой, поэтому к моменту написания этой книги доступно не так много моделей, но я думаю, что он станет популярным способом распространения и загрузки моделей уже к концу года. Все модели, указанные в этой главе, можно загрузить через репозиторий `pytorch/vision` в PytorchHub и свободно использовать этот процесс загрузки вместо `torchvision.models`.

Заключение

В этой главе вы познакомились с принципами работы нейронных сетей на основе CNN, включая такие функции, как `Dropout`, `MaxPool` и `BatchNorm`. Вы также узнали о самых популярных современных архитектурах. Прежде чем перейти к следующей главе, поиграйте с архитектурами, о которых мы говорили, и сравните их между собой. (Помните, что обучать их не нужно! Просто скачайте веса и протестируйте модель.)

На этом мы завершаем наш обзор систем компьютерного зрения, использующих предварительно обученные модели, и переходим к индивидуальному решению нашей проблемы распознавания кошек и рыбок с помощью *переноса обучения*.

Дополнительные источники

- AlexNet: «ImageNet Classification with Deep Convolutional Neural Networks», Alex Krizhevsky et al. (2012), https://oreil.ly/CsoFv
- VGG: «Very Deep Convolutional Networks for Large-Scale Image Recognition», Karen Simonyan and Andrew Zisserman (2014), https://arxiv.org/abs/1409.1556

- Inception: «Going Deeper with Convolutions», Christian Szegedy et al. (2014), https://arxiv.org/abs/1409.4842
- ResNet: «Deep Residual Learning for Image Recognition», Kaiming He et al. (2015), https://arxiv.org/abs/1512.03385
- NASNet: «Learning Transferable Architectures for Scalable Image Recognition», Barret Zoph et al. (2017), https://arxiv.org/abs/1707.07012

ГЛАВА 4

Перенос обучения и другие фокусы

Рассмотрев архитектуры в предыдущей главе, вы можете задаться вопросом, можно ли загрузить уже обученную модель и обучить ее еще больше. Ответ — да! Это невероятно высокоэффективная методика в сфере глубокого обучения, называемая *переносом обучения* (transfer learning), при которой сеть, обученная на выполнение одной задачи (например, ImageNet), адаптируется к другой (рыбка и кошка).

Для чего это? Оказывается, архитектура, обученная на ImageNet, уже очень много знает об изображениях, но совсем мало — о том, является объект кошкой или рыбкой (или же собакой или китом). Поскольку вы больше не начинаете работу с практически пустой нейронной сети, применяя перенос обучения, вы, скорее всего, будете тратить гораздо меньше времени на обучение *и* сможете получить намного меньший набор данных для обучения. Традиционные методы глубокого обучения требуют огромных объемов данных для получения хороших результатов. С помощью переноса вы можете создавать классификаторы на уровне человека с несколькими сотнями изображений.

Перенос обучения с помощью ResNet

Теперь нужно создать модель ResNet, как мы делали в главе 3, и просто вставить ее в существующий цикл обучения. И вы сможете это сделать! В модели ResNet нет ничего волшебного; она построена из уже знакомых вам строительных блоков. Однако это большая модель, и хотя вы увидите

некоторые улучшения данных по сравнению с базовой ResNet, вам потребуется много данных, чтобы убедиться, что обучающий сигнал поступает во все части архитектуры и в достаточной мере обучает их новым задачам классификации. Мы стараемся избегать использования большого количества данных в этом подходе.

Но вот в чем дело: мы не имеем дела с архитектурой, которая была инициализирована со случайными параметрами, как делали это раньше.

Предварительно обученная модель ResNet уже содержит в себе информацию для распознавания и классификации изображения. Зачем пытаться ее переобучить? Вместо этого мы *настраиваем* сеть. Мы немного меняем архитектуру, добавляя в конце новый сетевой блок, заменив стандартные линейные слои тысячной категории, которые обычно выполняют классификацию ImageNet. Затем мы *замораживаем* все существующие слои ResNet и при обучении обновляем только параметры в новых слоях, но по-прежнему принимаем активации замороженных слоев. Это позволяет нам быстро обучать новые слои, сохраняя информацию, которая уже содержится в предварительно обученных слоях.

Для начала давайте создадим предварительно обученную модель ResNet-50:

```
from torchvision import models
transfer_model = models.ResNet50(pretrained=True)
```

Далее нам нужно заморозить слои. Для этого есть простой способ: мы не даем накапливать градиенты, используя `requires_grad()`. Мы должны сделать это для каждого параметра сети, при этом PyTorch предоставляет метод `parameters()`, с помощью которого это выполняется довольно просто:

```
for name, param in transfer_model.named_parameters():
    param.requires_grad = False
```

Затем нужно заменить последний классификационный блок новым, который мы будем обучать определять, кто на картинке — кошка или рыбка. В этом примере мы заменим его на пару слоев `Linear`, а также `ReLU` и `Dropout`, но здесь у вас также могут быть дополнительные слои CNN.

Возможно, вы не захотите замораживать слои `BatchNorm` в модели, так как они будут обучены аппроксимировать среднее и среднеквадратичное отклонение набора данных, на котором первоначально была обучена модель, а не набор данных, на котором вы хотите выполнить настройку. Часть *сигнала* от ваших данных может в конечном итоге быть потеряна, так как `BatchNorm` *корректирует* ваши входные данные. Вы можете посмотреть на структуру модели и заморозить только те слои, которые не являются `BatchNorm`:

```
for name, param in transfer_model.named_parameters():
    if("bn" not in name):
        param.requires_grad = False
```

К счастью, определение реализации ResNet в PyTorch хранит конечный блок классификатора как переменную экземпляра, `fc`, поэтому все, что нам нужно сделать, — это заменить его нашей новой структурой (другие модели, предлагаемые PyTorch, используют либо `fc`, либо `classifier`, поэтому, возможно, вы захотите проверить определение в источнике, если пробуете его с другим типом модели):

```
transfer_model.fc = nn.Sequential(nn.Linear(transfer_model.fc.in_
features,500), nn.ReLU(),
nn.Dropout(), nn.Linear(500,2))
```

В предыдущем коде мы использовали переменную `in_features`, которая позволяет нам захватывать несколько активаций, входящих в слой (в нашем случае 2048). Вы также можете использовать `out_features` для обнаружения выходящих активаций. Это удобные функции, когда вы соединяете сети, например строите «кирпичики»; если входящие компоненты слоя не соответствуют исходящим компонентам предыдущего слоя, вы получите ошибку во время обработки.

Итак, мы возвращаемся к нашему циклу обучения и затем обучаем модель в обычном режиме. Вы должны увидеть некоторые большие скачки точности даже в течение нескольких эпох.

Перенос обучения — это ключевой метод повышения точности вашего приложения для глубокого обучения, но чтобы повысить производительность нашей модели, мы можем использовать множество других приемов. Давайте рассмотрим некоторые из них.

Вычисление скорости обучения

Из главы 2 вы наверняка помните, что я ввел такую концепцию, как *скорость обучения*, для нейронных сетей и упомянул, что это один из самых важных гиперпараметров, которые вы можете изменить, а затем указал на то, что вам нужно для него использовать, предложив довольно небольшое число и варианты для экспериментов с различными значениями. Что ж... Плохая новость заключается в том, что именно так многие находят оптимальную скорость обучения для своих архитектур, обычно с помощью метода под названием *сеточный поиск*, тщательно перебирая подмножества значений скорости обучения и сравнивая результаты с контрольным набором данных.

Это отнимает невероятно много времени, и хотя люди делают именно так, часто многие ошибаются. Например, значение скорости обучения, которое эмпирически было обнаружено для работы с оптимизатором Adam, равно 3e-4. Оно известно как константа Карпатого, после того как Андрей Карпатый (в настоящее время ведущий эксперт по разработке искусственного интеллекта в Tesla) написал об этом в 2016 году в Twitter (https://oreil.ly/WLw3q). К сожалению, мало людей прочитали этот твит: «Я просто хотел убедиться, что люди понимают, что это шутка». Самое смешное, что 3e-4 является значением, которое часто может дать хорошие результаты, поэтому в этой шутке есть доля правды.

С одной стороны, у вас медленный и утомительный поиск, с другой — смутные и малопонятные знания, полученные в результате работы над бесчисленными архитектурами. В какой-то момент вы просто начнете *чувствовать*, какой будет хорошая скорость обучения даже на самостоятельно сделанной нейронной сети. Существует ли какой-либо еще вариант, кроме этих двух крайностей?

К нашей радости, да! Хотя вы будете удивлены, как мало людей используют этот метод. Малоизвестная работа Лесли Смита, научного сотрудника военно-морской исследовательской лаборатории США, описывает подход к поиску подходящей скорости обучения.[1]

[1] См. «Cyclical Learning rates for Training Neural Networks», Leslie Smith (2015) (https://arxiv.org/abs/1506.01186).

Но так было только до тех пор, пока Джереми Ховард не рассказал об этой методике на своей лекции fast.ai, что, безусловно, привело к тому, что метод начал завоевывать популярность в сообществе глубокого обучения. Идея довольно проста: в течение эпохи начинайте с небольшой скорости обучения и увеличивайте до более высокой скорости обучения в каждой мини-партии, в итоге получая высокую скорость в конце эпохи. Рассчитайте потери для каждой скорости, а затем, взглянув на график, выберите скорость обучения, которая дает наибольшее снижение. Например, посмотрите на график на рис. 4.1.

В этом случае мы должны рассмотреть использование скорости обучения около 1e-2 (отмечено кружком), так как это примерно та точка, где градиент спуска самый крутой.

Рис. 4.1. Скорость обучения по отношению к потерям

Обратите внимание, что нужна не нижняя точка кривой, которая может быть очевидной, а точка, которая быстрее всего подходит к нижней точке кривой.

Вот упрощенная версия того, что делает библиотека fast.ai:

```
import math
def find_lr(model, loss_fn, optimizer, init_value=1e-8, final_value=10.0):
    number_in_epoch = len(train_loader) - 1
    update_step = (final_value / init_value) ** (1 / number_in_epoch)
    lr = init_value
    optimizer.param_groups[0]["lr"] = lr
    best_loss = 0.0
    batch_num = 0
    losses = []
    log_lrs = []
    for data in train_loader:
        batch_num += 1
        inputs, labels = data
        inputs, labels = inputs, labels
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)

        # Сбой при чрезмерных потерях

        if batch_num > 1 and loss > 4 * best_loss:
            return log_lrs[10:-5], losses[10:-5]

        # Запись лучшего результата потерь

        if loss < best_loss or batch_num == 1:
            best_loss = loss

        # Хранение значений

        losses.append(loss)
        log_lrs.append(math.log10(lr))

        # Выполняет обратный проход и оптимизирует

        loss.backward()
        optimizer.step()

        # Обновление lr для следующего шага и сохранение

        lr *= update_step
        optimizer.param_groups[0]["lr"] = lr
    return log_lrs[10:-5], losses[10:-5]
```

Здесь мы выполняем итерацию пакетов, обучая сеть почти в рутинном режиме; мы передаем входные данные через модель, а затем получаем потери от этого пакета. Записываем значение `best_loss` и сравниваем новые потери с ним. Если новая потеря более чем в четыре раза превышает значение `best_loss`, мы вылетаем из функции, возвращаясь к тому, что имеем

на настоящий момент (поскольку потеря стремится к бесконечности). В противном случае мы продолжаем добавлять потери и логи текущей скорости обучения и обновляем скорость обучения в рабочем порядке до максимальной скорости в конце цикла. График может быть изображен с помощью функции `matplotlib plt`:

```
logs,losses = find_lr()
plt.plot(logs,losses)
found_lr = 1e-2
```

Обратите внимание, что мы возвращаем фрагменты логов и потерь `lr`. Это делается потому, что первые фрагменты обучения и несколько последних (особенно если скорость обучения довольно быстро становится очень большой), как правило, не дают нам много информации.

Реализация в библиотеке fast.ai также включает в себя взвешенное сглаживание, поэтому на графике вы получаете плавные линии, тогда как этот фрагмент дает острые пики.

Помните, что, поскольку эта функция фактически обучает модель и связана с настройками скорости обучения оптимизатора, необходимо предварительно сохранить и перезагрузить модель, чтобы вернуться в состояние, в котором она находилась до того, как вы вызвали `find_lr()`, а также повторно инициализировать выбранный оптимизатор. Можно сделать это сейчас, передавая скорость обучения, которую вы определили исходя из графика!

Так мы получаем хорошую скорость обучения, но можем добиться еще большего успеха с помощью *дифференциальной скорости обучения*.

Дифференциальная скорость обучения

Мы использовали одинаковую скорость обучения ко всей модели. При обучении модели с нуля, возможно, в этом и есть смысл, но когда речь идет о переносе обучения, то обычно можно получить большую точность, если использовать кое-что другое: обучение разных групп слоев с разной скоростью. Ранее в этой главе мы заморозили все предварительно обученные слои модели и обучили только новый классификатор, но мы можем настроить некоторые из слоев, допустим, используемые нами модели ResNet.

Возможно, некоторое обучение слоев, предшествующих классификатору, сделает нашу модель немного точнее. Но так как эти предыдущие слои уже были обучены на наборе данных ImageNet, возможно, обучать их нужно меньше, чем наши новые уровни? PyTorch предлагает для этого простой способ. Давайте изменим оптимизатор для модели ResNet-50:

```
optimizer = optimizer.Adam([
{ 'params': transfer_model.layer4.parameters(), 'lr': found_lr /3},
{ 'params': transfer_model.layer3.parameters(), 'lr': found_lr /9},
], lr=found_lr)
```

Так мы задаем скорость обучения для `layer4` (непосредственно перед нашим классификатором) равной трети *найденной* скорости обучения и девятой части для `layer3`. Эта комбинация выявлена эмпирическим путем, но не стесняйтесь экспериментировать. Есть кое-что еще, о чем следует упомянуть. Как вы, возможно, помните, в начале этой главы мы заморозили все эти предварительно обученные слои. Было бы неплохо задать для них другую скорость обучения, но на данный момент обучение модели не будет касаться их вообще, потому что они не накапливают градиенты. Давайте это изменим:

```
unfreeze_layers = [transfer_model.layer3, transfer_model.layer4]
for layer in unfreeze_layers:
    for param in layer.parameters():
        param.requires_grad = True
```

Теперь, когда параметры в этих слоях снова принимают градиенты, дифференциальная скорость обучения будет применена, когда вы отладите модель. Обратите внимание, что при необходимости вы можете замораживать и размораживать части модели и выполнять дальнейшую отладку на каждом слое отдельно! Поскольку мы уже рассмотрели скорость обучения, давайте перейдем к другому аспекту обучения моделей: данные, которые мы им подаем.

Аугментация данных

Одна из самых страшных фраз, которые только можно услышать в науке о данных: «Черт, моя модель переобучена!». Как я уже говорил в главе 2, переобучение происходит в том случае, когда модель решает отразить данные, представленные в обучающем наборе, а не сгенерировать обобщенное решение. Часто приходится слышать, что конкретная модель *запомнила набор данных*, то есть модель выучила ответы и продолжила плохо обрабатывать рабочие данные.

Традиционным способом предотвращения этой проблемы является накопление большого количества данных. Чем больше данных будет видеть модель, тем более общее представление она получит о задаче, которую пытается решить. При рассмотрении задачи стягивания, если вы не дадите модели просто сохранять все ответы (перегружая ее память таким большим количеством данных), то она будет вынуждена *сжимать* входные данные и, следовательно, создавать решение, которое не сможет просто сохранить ответы внутри нее. На практике это работает хорошо, но предположим, что у нас есть только тысяча изображений и мы делаем перенос обучения. Что же делать в таком случае?

Одним из подходов, который мы можем использовать, является *аугментация данных*. Если у нас есть изображение, мы можем проделать с ним несколько манипуляций, которые должны предотвратить переобучение и сделать модель более общей. Рассмотрим изображения кошки Гельветики на рис. 4.2 и 4.3.

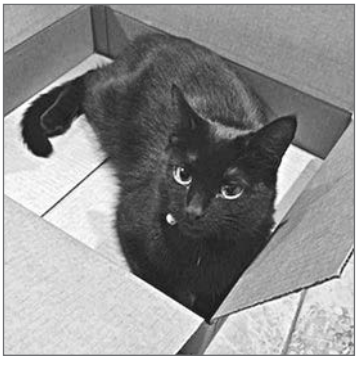

Рис. 4.2. Исходное изображение

Рис. 4.3. Перевернутая Гельветика

Для нас очевидно, что это один и тот же образ. Второе фото — это просто зеркальное отображение исходного изображения. Тензорное представление будет другим, так как RGB-значения будут находиться в разных местах трехмерного изображения. Но на фото все та же кошка, поэтому мы рассчитываем, что модель, обучаемая на этом изображении, научится распознавать форму кошки с левой или правой стороны кадра, а не будет просто связывать все изображение с *кошкой*. В PyTorch все просто. Возможно, вы помните этот фрагмент кода из главы 2:

```
transforms = transforms.Compose([
        transforms.Resize(64),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225] )
        ])
```

Он формирует конвейер преобразования, через который проходят все изображения при входе в модель для обучения. Но библиотека `torchivision.transforms` содержит много других функций преобразования, которые можно использовать для аугментации набора данных для обучения. Давайте рассмотрим некоторые из наиболее полезных и посмотрим, что происходит с Гельветикой при некоторых менее очевидных преобразованиях.

Преобразования Torchvision

В состав `torchvision` входит большой набор потенциальных преобразований, которые можно использовать для аугментации данных, а также два

способа создания новых преобразований. В этом разделе мы рассмотрим наиболее полезные из них, а также познакомимся с парой отдельных преобразований, которые вы можете использовать в своих собственных приложениях.

```
torchvision.transforms.ColorJitter(brightness=0, contrast=0, saturation=0,
                                   hue=0)
```

`ColorJitter` случайным образом меняет яркость, контрастность, насыщенность и оттенок изображения.

Для яркости, контраста и насыщенности вы можете задать либо число с плавающей точкой, либо кортеж с плавающей точкой, все неотрицательные числа в диапазоне от 0 до 1, и будет использоваться либо случайность между 0 и заданным значением с плавающей точкой, либо кортеж для генерации случайности между заданной парой значений с плавающей точкой. Для оттенка требуется значение с плавающей точкой или кортеж с плавающей точкой от –0,5 до 0,5, он будет генерировать случайные корректировки оттенка от [*-hue*,*hue*] или [*min*, *max*] (см. рис. 4.4).

Два преобразования случайным образом отражают изображение по горизонтальной или вертикальной оси:

```
torchvision.transforms.RandomHorizontalFlip(p=0.5)
torchvision.transforms.RandomVerticalFlip(p=0.5)
```

Либо задайте значение с плавающей точкой от 0 до 1 для вероятности отражения, либо по умолчанию примите значение с вероятностью отражения 50 %. Вертикально перевернутая кошка показана на рис. 4.5.

Рис. 4.4. ColorJitter применяется при 0,5 для всех параметров

Рис. 4.5. Вертикально перевернутое изображение

`RandomGrayscale` — это аналогичный тип преобразования, за исключением того, что он случайным образом преобразует изображения в оттенках серого в зависимости от параметра *p* (обработка по умолчанию 10 %):

```
torchvision.transforms.RandomGrayscale(p=0.1)
```

`RandomCrop` и `RandomResizeCrop`, как вы можете предполагать, обрезают картинку случайным образом, размер может быть либо переменной `int` для высоты и ширины, либо кортежем, содержащим разную высоту и ширину. На рис. 4.6 показан пример работы `RandomCrop`.

```
torchvision.transforms.RandomCrop(size, padding=None,
pad_if_needed=False, fill=0, padding_mode='constant')
torchvision.transforms.RandomResizedCrop(size, scale=(0.08, 1.0),
ratio=(0.75, 1.3333333333333333), interpolation=2)
```

Теперь нужно быть внимательнее, потому что если кадрированное изображение будет слишком маленьким, вы рискуете вырезать важные детали и модель выучит что-то неверно. Например, если на картинке изображена кошка, которая играет на столе, программа обрежет кошку и просто оставит часть стола, который будет классифицирован как *кошка*. Так себе результат. `RandomResizeCrop` изменит размер кадрирования, чтобы заполнить заданный размер, `RandomCrop` может сделать кадрирование близко к краю и в темных местах за пределами изображения.

Как вы уже знаете из главы 3, мы можем добавить отступ, чтобы сохранить необходимый размер изображения. По умолчанию отступ является

константой и заполняет пустые пикселы за пределами изображения тем значением, которое задано в параметре `fill`. Тем не менее советую использовать вместо него отступ `reflect`, поскольку практика показывает, что он работает немного лучше, а не просто заполняет пустое константное пространство.

`RandomResizeCrop` использует билинейную интерполяцию, но вы также можете выбрать ближайшую соседнюю или бикубическую интерполяцию, изменив параметр интерполяции. Более подробно см. на странице фильтров PIL.

Если вы хотите повернуть изображение случайным образом, `RandomRotation` будет варьироваться между `[-degrees, degrees]`, если `degrees` — это одно число с плавающей точкой или `int`, либо `(min,max)`, если это кортеж:

```
torchvision.transforms.RandomRotation(degrees, resample=False,expand=False,
                                      center=None)
```

Если для `expand` выбрано значение `True`, эта функция расширит выходное изображение так, чтобы оно могло включать весь поворот; оно обрезается по размерам входных данных по умолчанию.

Вы можете задать фильтр передискретизации PIL и при желании ввести `(x,y)` кортеж для центра вращения; в противном случае преобразование будет вращаться вокруг центра изображения.

Рисунок 4.7 — преобразование `RandomRotation` с заданными 45 градусами.

Рис. 4.6. RandomCrop. Размер=100

Рис. 4.7. RandomRotation. Градусы = 45

`Pad` — это универсальное преобразование отступа, которое добавляет к границам изображения отступы (дополнительную высоту и ширину):

```
torchvision.transforms.Pad(padding, fill=0, padding_mode=constant)
```

Отдельное значение в параметре `padding` будет применять отступ на эту длину во всех направлениях. Параметр `padding` из двух кортежей создаст отступ по длине (влево/вправо, сверху/снизу), а из четырех кортежей — отступ (слева, сверху, справа, снизу). По умолчанию отступ настроен на режим `constant`, который копирует значение `fill` в слоты отступа. Среди других вариантов можно найти параметр `edge`, который добавляет последние значения края изображения к длине отступа; `reflect` отражает значения изображения (кроме края) к границе; и `symmetric`, представляющий собой тот же `reflection`, но включающий в себя последнее значение изображения на краю. На рис. 4.8 показан `padding`, заданный на 25, и `padding_mode`, заданный на `reflect`. Посмотрите, как повторяется изображение коробки по краям.

`RandomAffine` позволяет задавать случайные аффинные преобразования изображения (масштабирование, повороты, переносы и/или сдвиг или любую из комбинаций). На рис. 4.9 показан пример аффинного преобразования.

```
torchvision.transforms.RandomAffine(degrees, translate=None, scale=None,
shear=None, resample=False, fillcolor=0)
```

Рис. 4.8. Оступ, padding = 25 и padding_mode = reflect

Параметр `degrees` — это либо отдельное число с плавающей точкой, либо `int`, либо кортеж. Если это отдельное число, то он производит случайные повороты между (`-degrees`, `degrees`), если кортеж — то случайные повороты между (`min, max`). Параметр `degrees` должен быть точно задан, чтобы предотвратить вращение, в данном случае настроек по умолчанию нет. Параметр `translate` — это кортеж из двух множителей (*horizontal_multipler*, *vertical_multiplier*).

Во время преобразования горизонтальный сдвиг, `dx`, дискретизируется в диапазоне $-image\_width \times horizontal\_multiplier < dx < img\_width \times horizontal\_width$, а вертикальный сдвиг дискретизируется таким же образом в отношении высоты изображения и вертикального множителя.

Масштабирование обрабатывается другим кортежем (`min, max`), а коэффициент равномерного масштабирования дискретизируется случайным образом. Сдвиг может быть либо отдельным числом с плавающей точкой/параметром `int`, либо кортежем и дискретизирует случайным образом так же, как и параметр `degrees`. Наконец, `resample` позволяет вам при необходимости создать фильтр передискретизации PIL, а `fillcolor` — это опциональный параметр `int`, задающий цвет заливки для областей внутри конечного изображения, которые находятся за пределами конечного преобразования. Что касается того, какие преобразования вы должны использовать в конвейере аугментации данных, для начала я определенно рекомендую использовать различные случайные отражения, колебание цветового тона, вращение и кадрирование.

Рис. 4.9. RandomAffine на 10 градусов, сдвиг = 50

В `torchvision` доступны и другие преобразования; ознакомиться с более подробной информацией можно в документации. Но вы, возможно, захотите создать особое преобразование специально под вашу область данных, которое не включено в документацию, поэтому, как вы увидите далее, PyTorch предлагает различные способы формирования специализированных преобразований.

Цветовое пространство и лямбда-преобразование

Вас может удивить, но до сих пор мы работали с изображениями в довольно стандартной 24-битной модели RGB цветового пространства, где каждый пиксел имеет 8-битное красное, зеленое и синее значение для обозначения цвета этого пиксела. Впрочем, доступны и другие цветовые пространства!

Распространенной альтернативой является модель HSV, имеющая три 8-битных значения — *тон, насыщенность, значение*. Некоторые считают, что эта система моделирует зрение человека точнее, чем традиционное цветовое пространство RGB. Но почему это важно? Гора в модели RGB — это та же гора в модели HSV, так?

Результат работы глубокого обучения в области колоризации показывает, что другое цветовое пространство может давать чуть более высокую точность, чем модель RGB. Гора может оставаться горой, но тензор, который формируется в каждом представлении пространства, будет разным, и одно пространство может захватить что-то о ваших данных лучше, чем другое.

В сочетании с ансамблями легко можно создать ряд моделей, который объединяет результаты обучения на моделях RGB, HSV, YUV и цветовом пространстве LAB, чтобы получить еще несколько процентных точек точности из вашего конвейера прогнозирования.

Проблемка заключается в том, что PyTorch не предлагает преобразование, которое может сделать это. Однако он предоставляет несколько инструментов, которые мы можем использовать для случайного изменения изображения со стандартной модели RGB на модель HSV (или другое цветовое пространство). Во-первых, из документации PI понятно, что для перевода изображения PIL из одного цветового пространства в другое можно использовать `Image.convert()`.

Можно было бы написать пользовательский класс `transform` для выполнения этого преобразования, но PyTorch добавляет класс `transforms.Lambda`, чтобы мы могли легко обернуть любую функцию и сделать ее доступной для конвейера преобразования. Вот наша пользовательская функция:

```
def _random_colour_space(x):
    output = x.convert("HSV")
    return output
```

Затем она оборачивается в класс `transforms.Lambda` и может использоваться в любом стандартном конвейерном преобразовании:

```
colour_transform = transforms.Lambda(lambda x: _random_colour_space(x))
```

Ничего страшного, если мы хотим преобразовать *каждое* изображение в модель HSV, но на самом деле нам это не нужно. Нам бы хотелось, чтобы изображения в каждом пакете менялись случайным образом, поэтому вполне вероятно, что изображение будет представлено в разных цветовых пространствах в разных эпохах. Мы могли бы обновить нашу исходную функцию, чтобы сгенерировать случайное число и использовать его для генерации случайной вероятности изменения изображения, но мы пойдем ленивым путем и используем `RandomApply`:

```
random_colour_transform = torchvision.transforms.RandomApply(
                          [colour_transform])
```

`RandomApply` по умолчанию заполняет параметр `p` значением `0.5`, поэтому вероятность применения преобразования составляет 50/50. Поэкспери-

ментируйте с добавлением дополнительного цветового пространства и вероятностью применения трансформации, чтобы увидеть, как это повлияет на нашу задачу с кошкой и рыбкой.

Давайте посмотрим на более сложное пользовательское преобразование.

Пользовательские классы преобразования

Иногда простой лямбды недостаточно; может быть, у нас есть некоторая инициализация или состояние, которое требуется отслеживать. В этих случаях можно создать собственное пользовательское преобразование, которое работает либо с данными изображения PIL, либо с тензором. Такой класс должен реализовывать два метода: `__call__`, который конвейер преобразования будет вызывать в процессе преобразования; и `__repr__`, который должен возвращать строковое представление преобразования, а также любое состояние, которое может быть полезно для целей диагностики.

В следующем коде мы реализуем класс преобразования, который добавляет к тензору случайный гауссовский шум. Когда класс инициализируется, мы передаем среднее и нормальное распределение шума, который нам требуется, и во время использования метода `__call__` выбираем это распределение и добавляем его во входящий тензор:

```
class Noise():
    """Добавляет гауссов шум к тензору.

    >>> transforms.Compose([
    >>>     transforms.ToTensor(),
    >>>     Noise(0.1, 0.05)),
    >>> ])

    """
    def __init__(self, mean, stddev):

        self.mean = mean
        self.stddev = stddev
    def __call__(self, tensor):
        noise = torch.zeros_like(tensor).normal_(self.mean, self.stddev)
        return tensor.add_(noise)

    def __repr__(self):
        repr = f"{self.__class__.__name__ }(mean={self.mean},
               stddev={self.stddev})"
        return repr
```

Если мы добавим это в конвейер, то увидим результаты вызова метода `__repr__`:

```
transforms.Compose([Noise(0.1, 0.05)])
>> Compose(
    Noise(mean=0.1,sttdev=0.05)
)
```

Поскольку преобразования не имеют никаких ограничений и просто наследуются от базового класса объекта Python, можно делать что угодно. Хотите полностью заменить изображение во время выполнения программы какой-то из картинок Google? Запустить образ через совершенно другую нейронную сеть и передать результат по конвейеру? Применить серию преобразований изображения, которые превращают изображение в безумную отражающую тень? Все это можно и даже нужно. Было бы интересно посмотреть, ухудшит ли эффект *Twirl* из Photoshop точность или, наоборот, улучшит ее. Почему бы не попробовать?

Помимо преобразований, есть еще несколько способов выжать из модели как можно большую производительность. Давайте посмотрим на другие примеры.

Начните с меньшего и получите больше!

Такой совет может показаться странным, но он реально работает: начните с меньшего и получите больше. Я имею в виду, что если вы обучаетесь на изображениях 256 × 256, создайте еще несколько наборов данных, в которых изображения были бы масштабированы до 64 × 64 и 128 × 128. Создайте свою модель с набором данных 64 × 64, отладьте ее как обычно, а затем обучите *точно такую же модель* с набором данных 128 × 128. Не с нуля, а используя параметры, которые уже были обучены. Как только вы получите максимум из набора данных 128 × 128, перейдите к целевым данным 256 × 256. Скорее всего, вы найдете процентную точку или два улучшения в точности.

Хотя мы не знаем точно, почему это работает, гипотеза заключается в том, что, обучаясь при более низких разрешениях, модель узнает об общей структуре изображения и может обработать эти знания по мере увеличения изображений.

Это преимущество, если нужно выжать из модели максимум. Если вы не хотите, чтобы в хранилище болталось несколько копий набора данных, используйте преобразования `torchvision`. Делается это с помощью функции `Resize`:

```
resize = transforms.Compose([ transforms.Resize(64),
 …_other augmentation transforms_…
 transforms.ToTensor(),
 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
```

Но за это придется заплатить и потратить больше времени на обучение, так как PyTorch должен всякий раз применять изменение размера. Если вы заранее изменили размеры всех изображений, то, скорее всего, обучение пройдет быстрее за счет заполнения жесткого диска. Но разве так происходит не всегда?

Эта концепция «начинай с меньшего и получи больше» также применима к архитектуре. Использование архитектуры ResNet, такой как ResNet-18 или ResNet-34, для проверки подходов к преобразованиям и понимания того, как работает обучение, обеспечивает гораздо более полноценную обратную связь, чем при использовании модели ResNet-101 или ResNet-152. Начните с малого, продвигайтесь вперед, и в теории вы сможете повторно использовать меньшие прогоны модели при прогнозировании, добавляя их в ансамблевую модель.

Ансамбли

Что может быть лучше, чем одна модель прогнозирования? Ну а как насчет нескольких?

Ансамблирование — это методика, которая довольно распространена в более традиционных методах машинного обучения и в глубоком обучении работает так же хорошо. Идея состоит в том, чтобы получить прогноз из ряда моделей и совместить их для получения окончательного ответа. Поскольку разные модели будут иметь разные преимущества в разных областях, мы предполагаем, что сочетание всех их прогнозов даст более точный результат, чем одна модель.

Существует много подходов к ансамблям, но не будем их подробно описывать здесь. Предлагаю простой способ начать работу с ансамблями, дающий, по моему опыту, еще 1 % точности, — просто усредните прогнозы:

```
# При условии, что у вас есть список моделей в моделях,
# а вход – это ваш входной тензор

predictions = [m[i].fit(input) for i in models]
avg_prediction = torch.stack(b).mean(0).argmax()
```

Метод `stack` объединяет массив тензоров, поэтому если бы мы работали над решением задачи «кошка/рыбка», а в нашем ансамбле было четыре модели, мы бы получили тензор 4 × 2, построенный из четырех тензоров 1 × 2. А `mean` делает то, чего вы ожидаете: принимает среднее значение, хотя следует передать размерность 0, чтобы убедиться, что `mean` принимает среднее значение по первой размерности, вместо простого суммирования всех элементов тензора и получения скалярных выходных данных.

Наконец, как вы видели раньше, `argmax` выбирает индекс тензора с наивысшим элементом. Легко представить и более сложные подходы. Возможно, веса могут быть добавлены к прогнозу каждой отдельной модели, и эти веса могут быть скорректированы, если модель получает правильный или неправильный ответ. Какие модели использовать? Я обнаружил, что комбинации ResNet (например, 34, 50, 101) работают довольно хорошо, и ничто не мешает вам регулярно сохранять свою модель и использовать различные снимки модели в вашем ансамбле!

Заключение

Подходя к концу главы 4, мы прощаемся с изображениями, чтобы перейти к тексту. Надеюсь, что вы не только поняли, как сверточные нейронные сети работают с изображениями, но и узнали много полезных приемов, включая перенос обучения, определение скорости обучения, аугментацию данных и ансамблирование, которое вы можете использовать в своем конкретном приложении.

Дополнительные источники

Если вам интересно узнать больше об изображениях, ознакомьтесь с курсом fast.ai от Джереми Ховарда, Рэйчел Томас и Сильвен Гуггер. Как я уже упоминал, определение скорости обучения в этой главе — это упрощенная версия используемого ими метода. Этот курс содержит более подробную информацию о многих приемах из этой главы. Библиотека fast.ai, построенная на PyTorch, позволяет легко использовать отображения (и текст!).

- Документация Torchvision, https://oreil.ly/vNnST
- Документация PIL/Pillow, https://oreil.ly/Jlisb
- «Cyclical Learning Rates for Training Neural Networks», Leslie N. Smith (2015), https://arxiv.org/abs/1506.01186
- «ColorNet: Investigating the Importance of Color Spaces for Image Classification», Shreyank N. Gowda and Chun Yuan (2019), https://arxiv.org/abs/1902.00267

ГЛАВА 5

Классификация текста

Итак, давайте оставим в покое изображения и переключим внимание на другую область, где традиционные методы глубокого обучения значительно продвинулись вперед. Речь пойдет об *обработке естественного языка* (NLP). Хорошим примером является Google Translate. Изначально код, который обрабатывал перевод, представлял собой ощутимые 500 000 строк кода. Новая система на основе TensorFlow составляет примерно 500 и работает лучше, чем старый метод.

Недавние прорывы также произошли в сфере применения переноса обучения (о котором вы узнали в главе 4) к проблемам NLP. Новые архитектуры, такие как Transformer, привели к созданию сетей, подобных OpenAI's GPT-2, более крупный вариант которого производит текст, практически сопоставимый по качеству тексту, который может создать человек (на самом деле OpenAI не выпустил веса этой модели, опасаясь ее неправильного использования).

В этой главе мы кратко рассмотрим рекуррентные нейронные сети и вложения. Затем мы рассмотрим библиотеку `torchtext` и то, как ее можно использовать для обработки текста с моделью на основе LSTM.

Рекуррентные нейронные сети

Если мы вернемся к тому, как мы до сих пор использовали архитектуры на основе CNN, то увидим, что они всегда работали в один полный момент времени. Давайте рассмотрим два предложения:

```
The cat sat on the mat.

She got up and impatiently climbed on the chair, meowing for food.
```

Перевод:

```
Эта кошка сидела на коврике.
Она встала и, мяукнув, нетерпеливо запрыгнула на стул, требуя еды.
```

Допустим, вы должны были вставить эти два предложения, одно за другим, в CNN и спросить, *где кошка*?

У вас возникнут трудности, так как у сети нет понятия *памяти*. Это невероятно важно, если речь идет о данных с временным доменом (например, текст, речь, видео и временные данные).[1] *Рекуррентные нейронные сети* (RNN) решают эту проблему, давая нейронной сети память через *скрытое состояние*.

Что представляет собой RNN? Мне нравится такое объяснение: «Представьте себе нейронную сеть, пересекающуюся с циклом `for`». На рис. 5.1 показана схема классической структуры RNN.

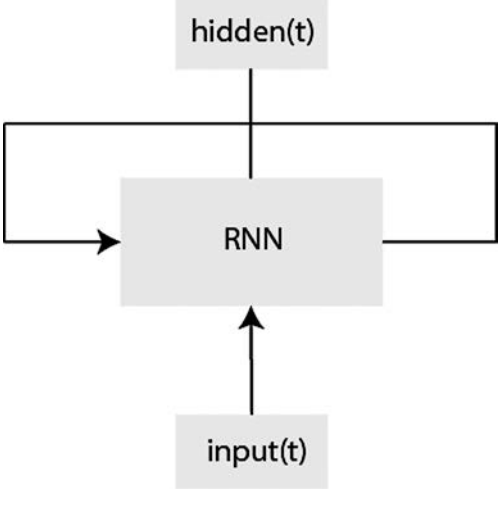

Рис. 5.1. RNN

[1] Обратите внимание, что это не невозможно сделать с CNN; в последние несколько лет было проведено много исследований, посвященных применению сетей на основе CNN во временном домене. Мы не будем их здесь рассматривать. Более подробную информацию ищите в «Temporal Convolutional Networks: A Unified Approach to Action Segmentation» (2016), Коллин Ли. И seq2seq! (https://arxiv.org/abs/1608.08242).

Мы добавляем входные данные с временным шагом *t* и получаем *скрытое* состояние выхода *ht*, а выходные данные также возвращаются в RNN для следующего временного шага. Мы можем развернуть эту сеть, как показано на рис. 5.2, чтобы глубже рассмотреть ее.

Здесь имеется группировка полносвязных слоев (с общими параметрами), ряд входов и выход. Входные данные поступают в сеть, а следующий элемент в последовательности прогнозируется как выходной. В развернутом виде RNN можно рассматривать как конвейер полносвязных слоев с последовательным входным сигналом, поступающим на следующий слой в последовательности (с обычными нелинейностями между слоями, такими как ReLU). Когда у нас есть завершенная спрогнозированная последовательность, мы должны выполнить обратное распространение ошибок через RNN. Поскольку необходимо вернуться назад, такой процесс называется обратным распространением ошибок во времени. Ошибка вычисляется по всей последовательности, затем сеть разворачивается (см. рис. 5.2), и осуществляется расчет градиентов для каждого временного шага и объединение для обновления общих параметров сети. Представьте, что вы делаете обратное распространение в отдельных сетях и суммируете все градиенты.

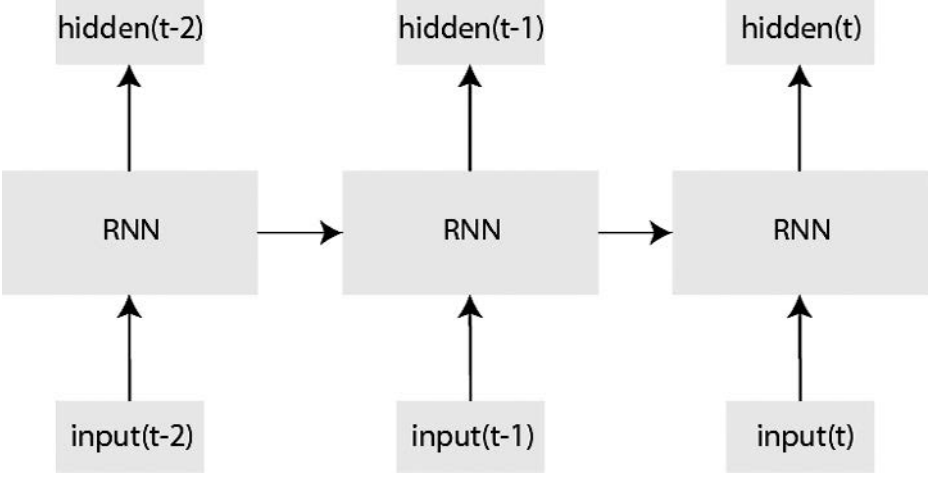

Рис. 5.2. Развернутая RNN

Это теория RNN. Но у этой простой структуры есть сложности, о которых нужно поговорить. Также мы затронем способы их устранения с помощью новых архитектур.

Сети с долгой краткосрочной памятью

На практике RNN особенно чувствительны к проблеме исчезающего градиента, о которой мы говорили в главе 2, или к потенциально худшему сценарию *взрывающихся градиентов*, где ошибка стремится к бесконечности. Ни в том ни в другом нет ничего хорошего, поэтому RNN не подходили для решения многих задач, которые изначально планировалось решать с их помощью. Все изменилось в 1997 году, когда Сепп Хохрайтер и Юрген Шмидхубер представили вариант RNN с долгой кратковременной памятью (LSTM).

На рис. 5.3 представлена диаграмма слоя LSTM. Я знаю, здесь много чего происходит, но сложно это только на первый взгляд. Серьезно!

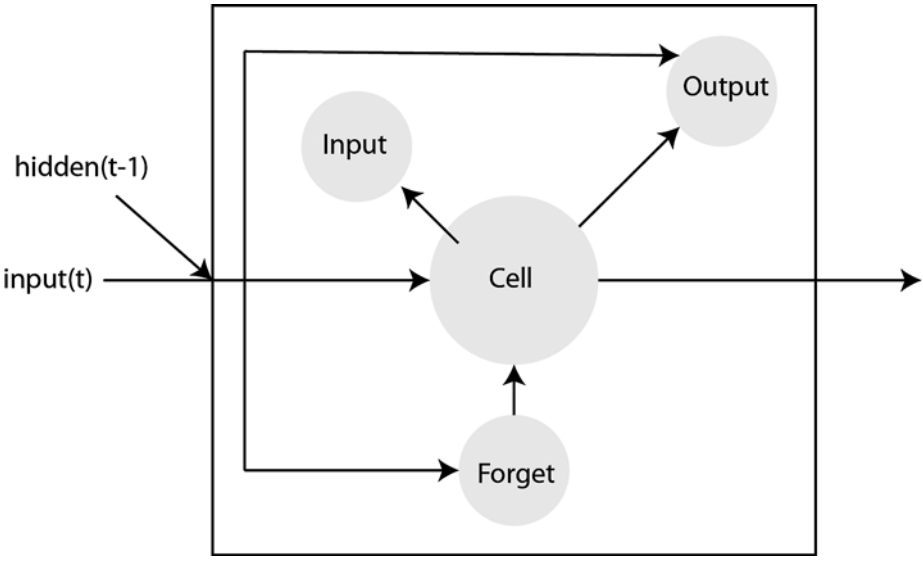

Рис. 5.3. LSTM

Выглядит довольно пугающе, согласен. Основное, о чем вам нужно помнить, — три вентиля (входной, выходной и забывания). В стандартной RNN мы «запоминаем» все навсегда. Но наш мозг (к сожалению!) работает не так. Вентиль забывания LSTM функционирует таким образом, что по мере продолжения нашей цепочки входных данных начало цепочки становится менее важным. LSTM забывает то, что сети узнают во время обучения, поэтому, если сети нужно быть очень забывчивой, параметры вентиля забывания это обеспечат.

Ячейка оказывается «памятью» сетевого слоя, а входной и выходной вентили и вентиль забывания будут определять, как данные проходят через слой. Данные могут просто проходить, они могут «записываться» в ячейку или могут (или не могут!) передаваться на следующий слой, модифицированный выходным вентилем.

Этого было достаточно для решения проблемы исчезающего градиента, кроме того, данный набор является полным по Тьюрингу, поэтому теоретически можно сделать любой расчет, который можно выполнить на компьютере, используя один из них.

Но, конечно же, это не все. Со временем в LSTM произошли кое-какие изменения, и мы рассмотрим некоторые из них в следующих разделах.

Управляемые рекуррентные блоки

С 1997 года было создано много вариантов базовой сети LSTM, в большинстве из которых вам, скорее всего, разбираться не обязательно. Однако стоит упомянуть об одном варианте, который появился в 2014 году, — управляемом рекуррентном блоке (GRU), поскольку он стал довольно популярным в некоторых сферах. На рис. 5.4 показана структура архитектуры GRU.

Главный вывод: в GRU вентиль забывания объединен с выходным вентилем. Это означает, что они имеют меньше параметров, чем LSTM, и поэтому склонны быстрее обучаться и использовать меньше ресурсов во время работы. Поэтому, а также потому, что они, по сути, заменяют LSTM, GRU стали довольно популярными. Хотя, откровенно говоря, слияние выходного вентиля и вентиля забывания делает их менее мощными, чем LSTM, поэтому в целом я рекомендую поэкспериментировать с GRU или LSTM в вашей сети и посмотреть, что работает лучше. Или просто примите

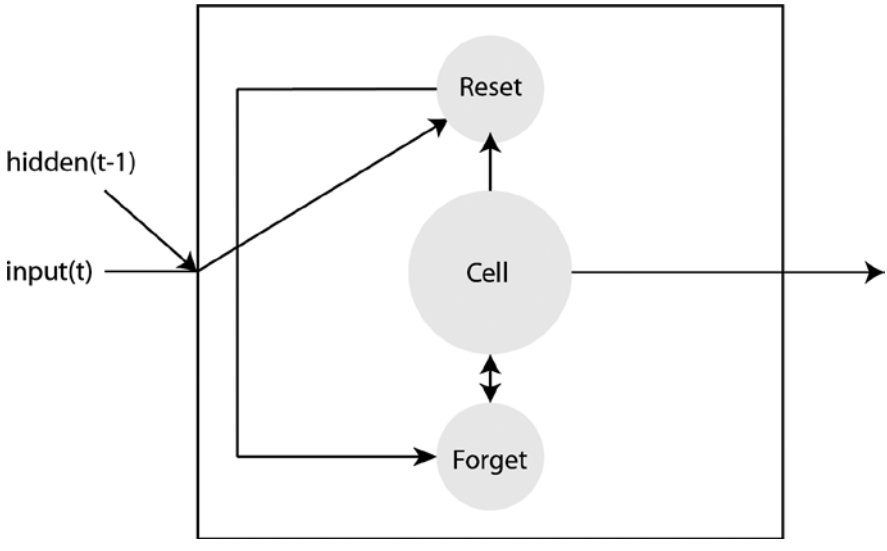

Рис. 5.4. GRU

за данность, что LSTM может обучаться немного медленнее, но в конечном итоге даст лучший результат. Вы не обязаны следовать последним веяниям моды — правда!

biLSTM

Другим распространенным вариантом LSTM является двунаправленный LSTM, или biLSTM. Как вы уже знаете, традиционные LSTM (и RNN в целом) могут «смотреть в прошлое», поскольку уже обучены и принимают решения. К сожалению, иногда нужно еще и «смотреть в будущее».

Это особенно актуально в таких приложениях, как перевод и распознавание рукописного текста, когда для определения выходных данных то, что следует за текущим состоянием, может быть столь же важным, как и предыдущее состояние.

biLSTM решает эту проблему самым простым способом: это, по сути, две стековые LSTM, причем входные данные отправляются в прямом направлении в одной LSTM, а во второй — в обратном. На рис. 5.5 показано, как biLSTM работает через входные данные в двух направлениях для получения выходных данных.

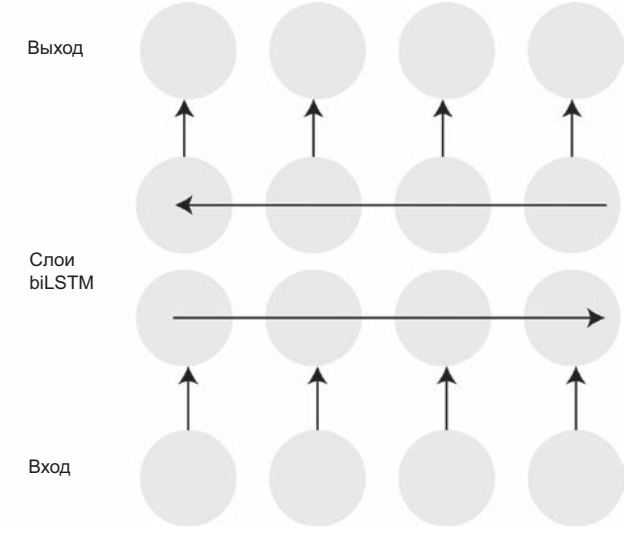

Рис. 5.5. biLSTM

Как вы позже узнаете из этой главы, PyTorch позволяет легко создавать biLSTM, передавая параметр `bidirectional=True` при создании блока `LSTM()`.

На этом мы завершаем наш тур по архитектуре на основе RNN. В главе 9 мы вернемся к вопросу архитектуры, когда будем рассматривать модели BERT и GPT-2 на базе Transformer.

Вложения (Embeddings)

Мы уже готовы писать код! Но прежде чем сделать это, подумайте: как мы представляем слова в сети? Мы подаем тензоры чисел в сеть и выводим их. Работая с изображениями, весьма очевидно конвертировать их в тензоры, представляющие красный/зеленый/синий параметры компонентов, и они уже естественным образом воспринимаются как массивы, поскольку имеют заданную высоту и ширину. А как же слова? Предложения? Как это работает?

Самый простой подход остается тем же, что вы найдете во многих подходах NLP, и он называется *горячим кодированием* (one-hot coding).

Он довольно простой! Давайте посмотрим на наше первое предложение из начала главы:

```
The cat sat on the mat.
```

Если учесть, что это весь словарь нашего мира, у нас есть следующий тензор: `[the, cat, sat, on, mat]`. Горячее кодирование просто означает, что мы создаем вектор размером со словарь и для каждого слова в нем выделяем вектор с одним параметром, равным 1, а остальные — 0:

```
the – [1 0 0 0 0]
cat – [0 1 0 0 0]
sat – [0 0 1 0 0]
on  – [0 0 0 1 0]
mat – [0 0 0 0 1]
```

Теперь мы конвертировали слова в векторы и можем подавать их в нашу сеть. Кроме того, мы можем добавить дополнительные символы в наш словарь, например UNK (неизвестно) — для слов, не входящих в словарь, и START/STOP, чтобы обозначить начало и конец предложений.

Горячее кодирование имеет несколько ограничений, которые станут понятнее, если мы добавим в наш словарь еще одно слово: *котенок* (*kitty*). Исходя из нашей схемы кодирования, *котенок* будет представлен как `[0 0 0 0 0 1]` (со всеми другими векторами с нулями).

Во-первых, вы можете видеть, что если мы хотим смоделировать реальный набор слов, наши векторы будут очень длинными, лишенными информации. Во-вторых, и, возможно, что еще более важно, мы знаем, что слова «котенок» и «кошка» очень *тесно взаимосвязаны* (то же самое и с сочетанием «черт побери!», но оно, к счастью, не входит в наш словарь!) и их невозможно представить с помощью горячего кодирования; эти два слова совершенно разные.

Подход, который стал более популярным в последнее время, заключается в замене горячего кодирования на матрицу вложения (*embedding matrix*) (горячее кодирование — это и есть непосредственно сама матрица вложения, которая не содержит никакой информации о взаимосвязи между словами). Суть в том, чтобы сократить размерность векторного пространства до некоторой более управляемой степени и использовать само пространство.

Например, если у нас есть вложение в двумерное пространство, возможно, что *кошка* может быть представлена тензором `[0.56, 0.45]`, а *котенок* — `[0.56, 0.445]`, тогда как *коврик* может быть представлен тензором `[0.2, -0.1]`. Мы группируем подобные слова в векторном пространстве и можем проверить расстояние, например, с помощью евклидовой функции или косинусной функции расстояния, чтобы определить, насколько слова близки друг к другу. А как мы определяем, где слова попадают в векторное пространство? Слой вложения ничем не отличается от любого другого слоя, который вы видели до сих пор при построении нейронных сетей; мы инициализируем векторное пространство случайным образом, и процесс обучения обновляет параметры так, чтобы похожие слова или понятия притягивались друг к другу.

Известный пример встраивания векторов — алгоритм *word2vec*, выпущенный Google в 2013 году.[1] Это был набор векторного представления слов, обученных с использованием малослойной нейронной сети, показавший, что преобразование в векторное пространство улавливает концепции, лежащие в основе слов. Возьмем широко известный пример: если вы вытащите векторы для слов *король*, *мужчина* и *женщина* (*King*, *Man* и *Woman*), а затем вычтете вектор для слова *мужчина* (*Man*) из слова *король* (*King*) и добавите вектор женщина (*Woman)*, то вы получите результат, который будет векторным представлением для слова *королева* (*Queen*). Начиная с *word2vec*, стали доступны другие предварительно обученные вложения, такие как *ELMo*, *GloVe* и *fasttext*.

Что касается использования вложений в PyTorch, тут все просто:

```
embed = nn.Embedding(vocab_size, dimension_size)
```

Сюда будет относиться тензор vocab_size x dimension_size, инициализированный случайным образом. Я предпочитаю считать, что это просто гигантский массив или справочная таблица. Каждое слово в вашем словаре индексирует элемент, который является вектором dimension_size, поэтому если мы вернемся к нашей кошке и ее эпическим приключениям на коврике, у нас получится нечто подобное:

```
cat_mat_embed = nn.Embedding(5, 2)
cat_tensor = Tensor([1])
```

[1] См. «Efficient Estimation of Word Representations in Vector Space», Tomas Mikolov et al. (2013) (https://arxiv.org/abs/1301.3781).

```
cat_mat_embed.forward(cat_tensor)
```

```
> tensor([[ 1.7793, -0.3127]], grad_fn=<EmbeddingBackward>)
```

Мы создаем наше вложение, тензор, который содержит положение *кошки* в нашем словаре, и пропускаем его через `forward()` слоя. Так мы получаем рандомное вложение. Результат также указывает на то, что у нас есть градиентная функция, которую можно использовать для обновления параметров после ее объединения с функцией потерь.

Теперь мы знаем всю теорию и можем переходить к практике!

torchtext

Как и torchvision, PyTorch предоставляет официальную библиотеку torchtext для работы с конвейерами обработки текста. Однако torchtext проверен на практике не так хорошо, как torchvision, да и внимания к нему приковано гораздо меньше. А это значит, что он не так прост в использовании и не так широко известен. Тем не менее он остается огромной библиотекой, которая может выполнять большую часть рутинной работы по созданию текстовых наборов данных, поэтому мы будем использовать ее до конца главы.

Установить torchtext очень просто. Используйте любой стандартный `pip`:

```
pip install torchtext
```

или специализированный `conda` канал:

```
conda install -c derickl torchtext
```

Если в вашей системе нет *spaCy* (библиотека NLP) и pandas, установите их (опять же, с использованием `pip` или `conda`). *spaCy* используется для обработки текста в конвейере torchtext, а pandas — для изучения и очистки данных.

Получение наших данных: твиты!

В этом разделе мы будем строить модель сентимент-анализа, то есть анализ тональности текста, поэтому нам понадобится набор данных. Библиотека torchtext предоставляет набор встроенных наборов данных через модуль `torchtext.datasets`, но мы поработаем над ним с нуля, чтобы прочувствовать весь процесс создания собственного набора данных и передать его в созданную модель. Возьмем набор данных Sentiment140 dataset (http://help.sentiment140.com/for-students). Он создан на основе твитов, где каждый отрицательный твит оценивается как 0, нейтральный — 2 и 4 — положительный.

Скачайте zip-архив и распакуйте его. Мы будем работать с файлом training.1600000.processed.noemoticon.csv. Давайте посмотрим на файл с помощью pandas:

```
import pandas as pd
tweetsDF = pd.read_csv("training.1600000.processed.noemoticon.csv",
                       header=None)
```

В этот момент вы можете увидеть такую ошибку:

```
UnicodeDecodeError: 'utf-8' codec can't decode bytes in
position 80-81: invalid continuation byte
```

Поздравляю, теперь вы настоящий специалист по данным и можете заняться их очисткой! Из сообщения об ошибке видно, что используемому по умолчанию синтаксическому анализатору CSV на основе C, который использует pandas, не нравится что-то из Юникода (Unicode) в файле, поэтому нам нужно переключиться на анализатор на основе Python:

```
tweetsDF = pd.read_csv("training.1600000.processed.noemoticon.csv",
engine="python", header=None)
```

Давайте посмотрим на структуру данных, отобразив первые пять строк:

```
>>> tweetDF.head(5)
0  0  1467810672 ... NO_QUERY scotthamilton is upset that ...
1  0  1467810917 ... NO_QUERY      mattycus @Kenichan I dived many times ...
2  0  1467811184 ... NO_QUERY       ElleCTF   my whole body feels itchy
3  0  1467811193 ... NO_QUERY        Karoli @nationwideclass no, it's ...
4  0  1467811372 ... NO_QUERY      joy_wolf @Kwesidei not the whole crew
```

К сожалению, у нас нет поля заголовка в этом CSV (опять же, добро пожаловать в мир данных!), но, посмотрев веб-сайт и включив интуицию, мы можем увидеть, что нас интересуют именно последний столбец (текст твита) и первый столбец (наша маркировка). Тем не менее ярлыки не идеальны, поэтому давайте немного над этим поработаем. Давайте посмотрим, что мы имеем в нашем наборе данных для обучения:

```
>>> tweetsDF[0].value_counts()
4    800000
0    800000
Name: 0, dtype: int64
```

Любопытно, что в наборе данных для обучения нет нейтральных значений. Это означает, что мы могли бы обозначить задачу как двоичный выбор между 0 и 1 и выработать прогнозы на основании него, но сейчас мы придерживаемся первоначального плана, согласно которому в будущем у нас могут появиться нейтральные твиты. Чтобы кодировать классы как числа, начинающиеся с 0, мы сначала создаем столбец типа `category` из столбца маркировки:

```
tweetsDF["sentiment_cat"] = tweetsDF[0].astype('category')
```

Затем мы кодируем эти классы в виде числовой информации в другом столбце:

```
tweetsDF["sentiment"] = tweetsDF["sentiment_cat"].cat.codes
```

Затем мы сохраняем измененный CSV обратно на диск:

```
tweetsDF.to_csv("train-processed.csv", header=None, index=None)
```

Советую вам сохранить еще один CSV с небольшой выборкой из 1,6 миллиона твитов, чтобы вы тоже могли их протестировать:

```
tweetsDF.sample(10000).to_csv("train-processed-sample.csv", header=None,
    index=None)
```

Теперь нам нужно сообщить torchtext, что, по нашему мнению, является важным для целей создания набора данных.

Определение полей

Библиотека torchtext генерирует набор данных простым способом: вы говорите, что вам нужно, и она обрабатывает исходный CSV (или JSON). Для этого сначала определяем *поля*. Класс `Field` имеет значительное количество параметров, которые могут быть ему присвоены, и хотя вы, скорее всего, не будете использовать их все сразу, в табл. 5.1 представлена полезная информация о том, что можно делать с полем.

Таблица 5.1. Типы параметров поля

Параметр	Описание	Значение по умолчанию
`sequential`	Представляет ли поле последовательные данные (то есть текст). Если задано значение `False`, токенизация не применяется	`True`
`use_vocab`	Включить ли объект `Vocab`. Если задано значение `False`, поле должно содержать числовые данные	`True`
`init_token`	Токен, который будет добавлен в начало этого поля, чтобы указать начало данных	`None`
`eos_token`	Токен в конце предложения, добавляемый в конец каждой последовательности	`None`
`fix_length`	Если задано целое число, все записи будут выровнены до этой длины. Если `None`, длины последовательности будут гибкими	`None`
`dtype`	Тип тензорной партии	`torch.long`
`lower`	Переводит последовательность в нижний регистр	`False`
`tokenize`	Функция, которая будет выполнять токенизацию последовательности. Если установлено значение `spacy`, будет использоваться токенизатор `spaCy`	`string.split`
`pad_token`	Токен, который будет использоваться как отступ	`<pad>`
`unk_token`	Токен, который будет использоваться для обозначения слов, которых нет в `Vocabdict`	`<unk>`
`pad_first`	Отступ в начале последовательности	`False`
`trun`	Усечение в начале последовательности (при необходимости)	`False`
`cate_first`		

Как уже было отмечено, нас интересуют только маркировки и текст твитов. Мы определяем их, используя тип данных `Field`:

```
from torchtext import data

LABEL = data.LabelField()
TWEET = data.Field(tokenize='spacy', lower=true)
```

Мы определяем `LABEL` как `LabelField`, являющийся подклассом `Field`, который задает `sequential` в `False` (так как это наш класс числовой категории). `TWEET` — это стандартный объект `Field`, в котором мы решили использовать токенизатор spaCy и конвертировать весь текст в нижний регистр, но в другом случае мы используем значения по умолчанию, как показано в предыдущей таблице. Если при запуске этого примера этап построения словаря занимает очень много времени, попробуйте удалить параметр `tokenize` и выполнить повторный запуск. По умолчанию будет использовано простое разбиение строки на пробелы, что значительно ускорит шаг токенизации, хотя созданный словарь будет не таким хорошим, как тот, который создает spaCy.

После определения этих полей нам нужно создать список, который свяжет их со списком строк в CSV:

```
fields = [('score',None), ('id',None),('date',None),('query',None),
     ('name',None),
     ('tweet', TWEET),('category',None),('label',LABEL)]
```

Вооружившись объявленными полями, мы используем `TabularDataset`, чтобы применить это определение к CSV:

```
twitterDataset = torchtext.data.TabularDataset(
        path="training-processed.csv",
        format="CSV",
        fields=fields,
        skip_header=False)
```

Это может занять некоторое время, особенно с анализатором spaCy. Наконец, мы можем разделить наборы данных на обучение, тестирование и верификацию с помощью метода `split()`:

```
(train, test, valid) = twitterDataset.split(split_ratio=[0.8,0.1,0.1])

(len(train),len(test),len(valid))
> (1280000, 160000, 160000)
```

Вот пример, вытащенный из набора данных:

```
>vars(train.examples[7])

{'label': '6681',
 'tweet': ['woah',
  ',',
  'hell',
  'in',
  'chapel',
  'thrill',
  'is',
  'closed',
  '.',
  'no',
  'more',
  'sweaty',
  'basement',
  'dance',
  'parties',
  '?',
  '?']}
```

Удивительно, но случайным образом выбранный твит ссылается на закрытие клуба в Чапел-Хилл, который я часто посещал. Удастся ли вам найти что-то странное при погружении в данные?

Построение словаря

Традиционно на этом этапе мы бы создавали горячее кодирование каждого слова, которое присутствует в наборе данных, — довольно утомительный процесс. К счастью, `torchtext` сделает это за нас, а также позволит передать параметр `max_size`, чтобы ограничить словарь наиболее общеупотребительными словами. Обычно это делается для того, чтобы предотвратить создание огромной модели, требующей много памяти. В конце концов, мы не хотим перегружать наши графические процессоры. Давайте ограничим словарь в наборе данных для обучения максимумом в 20 000 слов.

```
vocab_size = 20000
TWEET.build_vocab(train, max_size = vocab_size)
```

Затем мы можем попросить объект экземпляра класса `vocab` дать некоторую информацию о нашем наборе данных. Сначала мы задаем стандартный вопрос: «Насколько большой наш словарь?».

```
len(TWEET.vocab)
> 20002
```

Cmon, cmon, что? Да, мы указали 20 000 слов, но torchtext по умолчанию добавит еще два специальных токена `<unk>` для неизвестных слов (например, тех, которые были отброшены в результате ограничения 20 000 слов, заданного с помощью `max_size`) и `<pad>`, токен отступа, который будет использован для подгонки всего текста примерно до одного размера, чтобы обеспечить эффективное пакетирование на GPU (помните, что его скорость зависит от работы на регулярных пакетах). Также можно задать символы `eos_token` или `init_token` при объявлении поля, но они не включены по умолчанию.

Теперь давайте посмотрим на наиболее общие слова в словаре:

```
>TWEET.vocab.freqs.most_common(10)
[('!', 44802),
 ('.', 40088),
 ('I', 33133),
 (' ', 29484),
 ('to', 28024),
 ('the', 24389),
 (',', 23951),
('a', 18366),
 ('i', 17189),
('and', 14252)]
```

Практически то, что вы и ожидали, поскольку мы не удаляем стоп-слова с помощью токенизатора spaCy. (Поскольку это всего 140 символов, если мы это сделаем, то наша модель может потерять слишком много информации).

Мы почти закончили с наборами данных. Теперь нужно создать загрузчик данных для подачи в цикл обучения. torchtext предоставляет метод `BucketIterator`, который будет производить то, что он называет `Batch`, — это немного похоже на загрузчик данных, который мы использовали на изображениях.

(Вскоре вы увидите, что нужно обновить цикл обучения, чтобы справиться с некоторыми странностями интерфейса `Batch`.)

```
train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
(train, valid, test),
batch_size = 32,
```

```
device = device)
```

Итак, вот полный код для компиляции наших наборов данных:

```
from torchtext import data

device = "cuda"
LABEL = data.LabelField()
TWEET = data.Field(tokenize='spacy', lower=true)

fields = [('score',None), ('id',None),('date',None),('query',None),
      ('name',None),
      ('tweet', TWEET),('category',None),('label',LABEL)]

twitterDataset = torchtext.data.TabularDataset(
                  path="training-processed.csv",
        format="CSV",
        fields=fields,
        skip_header=False)

(train, test, valid) = twitterDataset.split(split_ratio=[0.8,0.1,0.1])

vocab_size = 20002
TWEET.build_vocab(train, max_size = vocab_size)

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
(train, valid, test),
batch_size = 32,
device = device)
```

После сортировки обработки данных мы можем перейти к определению нашей модели.

Создание модели

Чтобы построить простую модель для классификации твитов в PyTorch, мы используем модули `Embedding` и `LSTM` (о них говорилось в первой половине этой главы):

```
import torch.nn as nn

class OurFirstLSTM(nn.Module):
    def __init__(self, hidden_size, embedding_dim, vocab_size):
        super(OurFirstLSTM, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.encoder = nn.LSTM(input_size=embedding_dim,
                hidden_size=hidden_size, num_layers=1)
```

```
        self.predictor = nn.Linear(hidden_size, 2)

    def forward(self, seq):
        output, (hidden,_) = self.encoder(self.embedding(seq))
        preds = self.predictor(hidden.squeeze(0))
        return preds

model = OurFirstLSTM(100,300, 20002)
model.to(device)
```

Все, что мы делаем в этой модели, — это создаем три слоя. Сначала слова в наших твитах помещаются в слой `Embedding`, который мы задали как вложение 300-мерного вектора. Затем он передается в `LSTM` со 100 скрытыми функциями (опять же, мы сжимаем 300-мерные входные данные, как делали это с изображениями). Наконец, выход LSTM (окончательное скрытое состояние после обработки входящего твита) проходит через стандартный полносвязный слой с тремя выходами, чтобы соответствовать нашим трем возможным классам (отрицательный, положительный или нейтральный). Далее мы переходим к циклу обучения!

Обновление цикла обучения

Ввиду некоторых особенностей torchtext нужно написать незначительно измененный цикл обучения. Сначала мы создаем оптимизатор (как обычно используем Adam) и функцию потерь. Поскольку у нас есть три потенциальных класса для каждого твита, мы используем `CrossEntropyLoss()` в качестве функции потерь. Однако оказывается, что в наборе данных есть всего два класса; если бы мы предположили, что их будет только два, то мы фактически могли бы изменить выходные данные модели, чтобы получить отдельное число от 0 до 1, а затем использовать потерю бинарной кросс-энтропии (BCE). Можно объединить сигмоидальный слой, который вмещает выходное значение в интервале от 0 до 1 плюс слой BCE, в одну функцию потерь PyTorch, `BCEWithLogitsLoss()`. Я говорю об этом, потому что если вы пишете классификатор, который всегда должен быть одним или другим состоянием, это лучше, чем стандартная потеря кросс-энтропии, которую мы собираемся использовать.

```
optimizer = optim.Adam(model.parameters(), lr=2e-2)
criterion = nn.CrossEntropyLoss()

def train(epochs, model, optimizer, criterion, train_iterator,
          valid_iterator):
```

```
    for epoch in range(1, epochs + 1):

        training_loss = 0.0
        valid_loss = 0.0
        model.train()
        for batch_idx, batch in enumerate(train_iterator):
            opt.zero_grad()
            predict = model(batch.tweet)
            loss = criterion(predict,batch.label)
            loss.backward()
            optimizer.step()
            training_loss += loss.data.item() * batch.tweet.size(0)
        training_loss /= len(train_iterator)


        model.eval()
        for batch_idx,batch in enumerate(valid_iterator):
            predict = model(batch.tweet)
            loss = criterion(predict,batch.label)
            valid_loss += loss.data.item() * x.size(0)

        valid_loss /= len(valid_iterator)
        print('Epoch: {}, Training Loss: {:.2f},
        Validation Loss: {:.2f}'.format(epoch, training_loss, valid_loss))
```

Главное, что нужно знать об этом новом цикле обучения, — это то, что мы должны ссылаться на `batch.tweet` и `batch.label`, чтобы получить конкретные поля, которые нас интересуют; они не так хорошо выпадают из нумератора, как в `torchvision`.

После того как мы обучили нашу модель с помощью этой функции, мы можем использовать ее для классификации некоторых твитов, чтобы выполнить простой сентимент-анализ, или анализ тональности.

Классификация твитов

Еще один недостаток torchtext заключается в том, что его сложно заставить что-либо прогнозировать. В этом случае вы можете эмулировать конвейер обработки, который осуществляется внутри, и сделать требуемый прогноз на выходе этого конвейера, как показано в этой небольшой функции:

```
def classify_tweet(tweet):
    categories = {0: "Negative", 1:"Positive"}
    processed = TWEET.process([TWEET.preprocess(tweet)])
    return categories[model(processed).argmax().item()]
```

Мы должны вызвать `preprocess()`, который выполняет токенизацию на основе spaCy. Затем мы можем вызвать `process()` для токенов в тензор, основываясь на уже построенном словаре. Единственное, на что следует обратить внимание, так это на то, что torchtext ожидает пакет строк, поэтому нужно преобразовать его в список списков, прежде чем передать его в функцию обработки. Затем мы подаем его в модель. Это создаст тензор, который выглядит следующим образом:

```
tensor([[ 0.7828, -0.0024]]
```

Тензорный элемент с наибольшим значением соответствует выбранному классу модели, поэтому мы используем `argmax()` для получения его индекса, а затем `item()` для преобразования этого тензора нулевой размерности в целое число Python, которое мы индексируем в нашем словаре `categories`.

Обучив нашу модель, давайте рассмотрим, как выполнить некоторые другие приемы, о которых вы уже знаете из глав 2–4.

Аугментация данных

Возможно, вам интересно, как именно можно выполнить аугментацию текстовых данных. В конце концов, нельзя же перевернуть текст горизонтально, как изображение! Но можно использовать некоторые другие методы работы с текстом, которые предоставят модели немного больше информации для обучения. Во-первых, можно заменить слова в предложении синонимами, например, фраза:

```
Эта кошка сидела на коврике.
```

может стать

```
Эта кошка сидела на подстилке.
```

Помимо мнения кошки о том, что подстилка намного мягче коврика, значение предложения не изменилось. Но *коврик* и *подстилка* будут сопоставлены в словаре в разных индексах, поэтому модель научится тому, что два предложения распределяются на одну и ту же маркировку и что

между этими двумя словами есть связь, поскольку все остальное в предложениях одинаково.

В начале 2019 года в работе «EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks» были предложены три другие стратегии аугментации: случайная вставка, случайная перестановка и случайное удаление. Давайте рассмотрим каждую из них.[1]

Случайная вставка

В случае с методом *случайной вставки* программа «смотрит» на предложение, а затем случайным образом вставляет синонимы существующих нонстоп-слов в предложение *n* раз. При том, что у вас есть способ получения синонима слова и способ исключения стоп-слов (общие слова, такие как *и, оно, артикль* и т. д.), показанных, но не реализованных в этой функции через `get_synonyms()` и `get_stopwords()`, реализация будет выглядеть следующим образом:

```
def random_insertion(sentence,n):
    words = remove_stopwords(sentence)
    for _ in range(n):
        new_synonym = get_synonyms(random.choice(words))
        sentence.insert(randrange(len(sentence)+1), new_synonym)
    return sentence
```

На практике пример, где происходит вставка к слову `кошка`, может выглядеть так:

```
The cat sat on the mat
The cat mat sat on feline the mat
```

Перевод:

```
Эта кошка сидела на коврике
Эта кошка коврик сидела на кошачьем этом коврике
```

[1] См. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks», Jason W. Wei and Kai Zou (2019) (https://arxiv.org/abs/1901.11196).

Случайное удаление

Как следует из названия, *случайное удаление* удаляет слова из предложения. Учитывая параметр вероятности p, оно пройдет через предложение и на основании этой случайной вероятности решит, нужно ли удалять слово или нет:

```
def random_deletion(words, p=0.5):
    if len(words) == 1:
        return words
    remaining = list(filter(lambda x: random.uniform(0,1) > p,words))
    if len(remaining) == 0:
        return [random.choice(words)]
    else
        return remaining
```

Реализация имеет дело с пограничными случаями — если есть только одно слово, метод возвращает его; и если мы в конечном итоге удалим все слова в предложении, метод выберет случайное слово из исходного набора.

Случайная перестановка

Аугментация в виде *случайной перестановки* берет предложение, а затем меняет слова внутри него *n* раз, при этом каждая итерация работает с ранее поменявшимся предложением. Реализация выглядит следующим образом:

```
def random_swap(sentence, n=5):
    length = range(len(sentence))
    for _ in range(n):
        idx1, idx2 = random.sample(length, 2)
        sentence[idx1], sentence[idx2] = sentence[idx2], sentence[idx1]
    return sentence
```

Мы выбираем два случайных числа на основе длины предложения, а затем просто продолжаем менять местами, пока не получим *n*.

Методы, представленные в EDA, в среднем улучшают точность примерно на 3 % при использовании небольшого количества маркированных примеров (примерно 500). Однако автор работы работы считает, что если в наборе данных содержится более 5000 примеров, то улучшение точности может упасть до 0,8 % или ниже, поскольку модель получает лучшее обобщение из большего объема доступных данных, по сравнению с улучшениями, которые может предоставить EDA.

Обратный перевод

Еще один популярный подход к аугментации наборов данных — *обратный перевод*. Он включает перевод предложения с целевого языка на один или несколько других языков, а затем обратный перевод на исходный язык. Для этого мы можем использовать библиотеку Python `googletrans`. Установите ее с помощью `pip`, поскольку на момент написания книги она недоступна в `conda`:

```
pip install googletrans
```

Теперь можно перевести наше предложение с английского на французский, а затем обратно на английский:

```
import googletrans
import googletrans.Translator

translator = Translator()

sentences = ['The cat sat on the mat']

translation_fr = translator.translate(sentences, dest='fr')
fr_text = [t.text for t in translations_fr]
translation_en = translator.translate(fr_text, dest='en')
en_text = [t.text for t in translation_en]
print(en_text)

>> ['The cat sat on the carpet']
```

Так мы получаем аугментированное предложение в переводе с английского на французский и обратно, но давайте сделаем еще один шаг и выберем язык случайным образом:

```
import random

available_langs = list(googletrans.LANGUAGES.keys())
tr_lang = random.choice(available_langs)
print(f"Translating to {googletrans.LANGUAGES[tr_lang]}")

translations = translator.translate(sentences, dest=tr_lang)
t_text = [t.text for t in translations]
print(t_text)

translations_en_random = translator.translate(t_text, src=tr_lang, dest='en')
en_text = [t.text for t in translations_en_random]
print(en_text)
```

В этом случае мы используем `random.choice`, чтобы получить случайный язык, перевести предложение на этот язык, а затем выполнить обратный перевод. Мы также передаем язык в параметр `src`, чтобы помочь Google Translate определить язык. Попробуйте и посмотрите, как это напоминает старую игру «Испорченный телефон».

Стоит знать о нескольких ограничениях. Во-первых, можно переводить только до 15 тысяч символов за раз, хотя это не должно быть слишком большой проблемой, если вы просто переводите предложения. Во-вторых, если вы собираетесь использовать его на большом наборе данных, нужно выполнить аугментацию данных в облачном экземпляре, а не на компьютере, потому что если Google запретит ваш IP, то вы не сможете использовать Google Translate в обычном режиме! Убедитесь, что отправляете несколько пакетов, а не весь набор данных одновременно. Это также должно позволить перезапустить пакеты переводов, если на сервере Google Translate произойдет ошибка.

Аугментация и torchtext

Возможно, вы заметили, что до сих пор при разговоре об аугментации я ни разу не упомянул torchtext. К сожалению, на то есть основания. В отличие от torchvision или torchau dio, torchtext не предлагает конвейер преобразования, а это немного грустно. Он предлагает способ выполнения предварительной и последующей обработки, но это работает только на уровне токена (слова), чего, возможно, и достаточно для замены синонимов, но это не обеспечивает достаточного контроля для таких операций, как обратный перевод. И если вы попытаетесь перехватить конвейеры для аугментации, то лучше сделать это в конвейере предварительной обработки, а не в конвейере постобработки, поскольку все, что вы увидите в нем, — это тензор из целых чисел, которые вы должны сопоставить со словами через правила словаря.

Поэтому предлагаю даже не тратить время на попытки использовать torchtext для аугментации данных. Лучше сделайте это не в PyTorch, используя такие методы, как обратный перевод, чтобы сгенерировать новые данные и передать их в модель, как если бы это были *реальные* данные.

На этом об аугментации всё, но прежде чем закончить главу, давайте посмотрим на слона, которого-то мы и не приметили.

Перенос обучения?

Возможно, вам интересно, почему мы не говорили о переносе обучения. В конце концов, это ключевой метод, который позволяет создавать точные модели на основе изображений, так почему же мы не можем применять его здесь? Что ж, оказалось, что получить перенос обучения, работающий на LSTM-сетях, немного сложнее. Но не невозможно. Мы вернемся к этой теме в главе 9, где вы узнаете, как настроить перенос обучения, работая с LSTM-сетями, и с Transformer-сетями.

Заключение

В этой главе мы рассмотрели конвейер обработки текстов, который охватывает кодирование и вложения, простую нейронную LSTM-сеть для выполнения классификации, а также некоторые стратегии аугментации текстовых данных. У вас есть широкое поле для экспериментов. Я решил перевести каждый твит в нижний регистр на этапе токенизации. В NLP это распространенный подход, но он отбрасывает важную информацию в твите. Фраза «Почему оно НЕ РАБОТАЕТ?» вызывает в нас больше эмоций, чем фраза «Почему оно не работает?», но мы отбросили это различие еще до того, как твиты попали в модель. Так что определенно попробуйте запустить модель с учетом регистра слева в токенизированом тексте. А еще попробуйте удалить стоп-слова из вашего входного текста, чтобы увидеть, повышается ли точность. Традиционные методы NLP склонны удалять их, но я обнаруживал, что методы глубокого обучения могут работать лучше, если стоп-слова оставить на входе (что мы и сделали в этой главе). Это происходит потому, что они дают модели больше контекста для обучения, в то время как сокращенным до важных слов предложениям может не хватать нюансов.

Можно изменить размер вектора вложения. Большие векторы означают, что вложение может захватывать больше информации о слове, которое оно моделирует, за счет использования большего количества памяти. Попробуйте перейти от 100- к 1000-мерному вложению и посмотрите, как это влияет на время и точность обучения.

Наконец, вы также можете поиграть с LSTM. Мы использовали простой подход, но можно увеличить `num_layers` для создания стековых LSTM,

увеличить или уменьшить количество скрытых признаков в слое или задать `bidirectional=true` для создания biLSTM.

Замена всего LSTM слоем GRU также будет интересным опытом — обучается ли сеть быстрее? Или точнее? Экспериментируйте и смотрите, что из этого получится!

А пока мы переходим из мира текстов в мир звуков с помощью `torchaudio`.

Дополнительные источники

- «Long Short-term Memory», S. Hochreiter and J. Schmidhuber (1997), https://oreil.ly/WKcxO
- «Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation», Kyunghyun Cho et al. (2014), https://arxiv.org/abs/1406.1078
- «Bidirectional LSTM-CRF Models for Sequence Tagging», Zhiheng Huang et al. (2015), https://arxiv.org/abs/1508.01991
- «Attention Is All You Need», Ashish Vaswani et al. (2017), https://arxiv.org/abs/1706.03762

ГЛАВА 6

Путешествие в мир звуков

Каждый день мы сталкиваемся с лучшими образцами глубокого обучения. Siri, или Google Now, или Alexa от Amazon — в основе их лежат нейронные сети. В этой главе мы познакомимся с библиотекой torchaudio PyTorch. Вы узнаете, как использовать ее при построении конвейера для классификации аудиоданных с помощью сверточной модели. Затем я предложу другой подход, который позволит использовать некоторые приемы, о которых вы узнали, читая о работе с изображениями, и получить хорошую точность, работая с набором аудиоданных ESC-50.

Но сначала давайте разберемся, что такое звук. Как он представляется в виде данных и какой тип нейронной сети нужно использовать, чтобы получить представление о наших данных?

Звук

Звук появляется в результате колебаний. Все звуки, которые мы слышим, представляют собой комбинации высокого и низкого давления, которые мы часто представляем в форме волн, как показано на рис. 6.1. На этом рисунке волна выше начала координат — это высокое давление, а волна ниже начала координат — низкое давление.

На рис. 6.2 показана более сложная форма волны полной песни.

В цифровом звуке мы *дискретизируем* эту форму волны много раз в секунду, обычно 44 100 для качества звучания для CD-диска, и сохраняем значения амплитуды волны в каждой точке семплирования. В момент времени t у нас хранится одно значение, в отличие от изображения, для хранения которого требуется два значения, x и y (для изображения в оттенках

серого). Если мы используем сверточные фильтры в нашей нейронной сети, нам нужен 1D-фильтр, а не 2D-фильтры, которые мы использовали для изображений.

Теперь, когда вы знаете о звуке чуть больше, давайте посмотрим на пример набора данных, чтобы вам было проще с ним познакомиться.

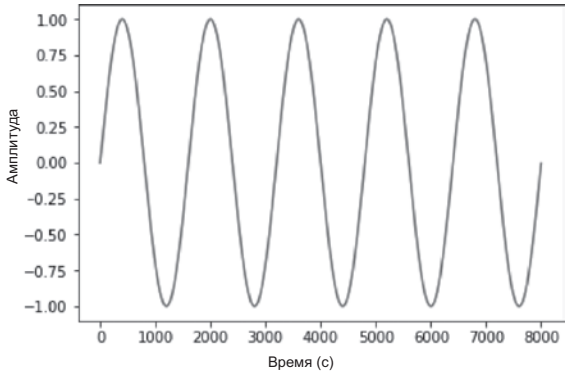

Рис. 6.1. Синусоидальная волна

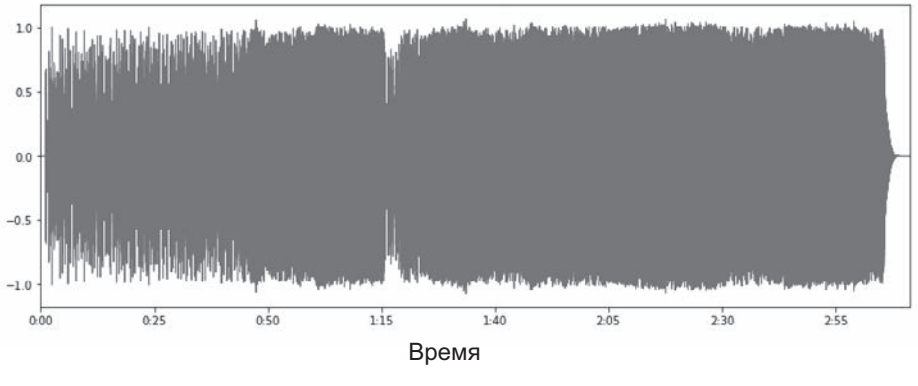

Рис. 6.2. Форма волны песни

Набор данных ESC-50

Набор данных системы классификации звуков окружающей среды (ESC) представляет собой набор полевых записей, каждая из которых длится 5 секунд и относится к одному из 50 классов (например, лай собаки, храп, стук в дверь). Мы будем использовать этот набор в дальнейшем, чтобы поэкспериментировать с двумя способами классификации аудиофайлов, а также изучить использование torchaudio для упрощения загрузки и работы с ними.

Получение набора данных

Набор данных ESC-50 представляет собой набор WAV-файлов. Вы можете скачать его, клонировав репозиторий Git:

```
git clone https://github.com/karoldvl/ESC-50
```

Или вы можете скачать весь репозиторий, просто используя команду `curl`:

```
curl https://github.com/karoldvl/ESC-50/archive/master.zip
```

Все WAV-файлы хранятся в *аудио*каталоге с такими именами файлов:

```
1-100032-A-0.wav
```

Нам важно последнее число в имени файла, потому что оно говорит нам, какому классу был присвоен этот файл звукозаписи. Другие части имени файла нам не важны, но в основном они связаны с набором данных Freesound, из которого были взяты данные ESC-50 (за одним исключением, к которому я скоро вернусь). Если вы хотите знать больше, ознакомьтесь с документом *README* в репозитории ESC-50.

Итак, когда мы загрузили набор данных, давайте посмотрим на звуки, которые он содержит.

Воспроизведение аудио в Jupyter

Если вы хотите прослушать звук в ESC-50, то вместо того, чтобы загружать файл в стандартный музыкальный плеер вроде iTunes, вы можете использовать встроенный аудиопроигрыватель Jupyter, `IPython.display.Audio`:

```
import IPython.display as display
display.Audio('ESC-50/audio/1-100032-A-0.wav')
```

Функция будет читать наши WAV- и MP3-файлы. Вы также можете генерировать тензоры, конвертировать их в массивы NumPy и воспроизводить их напрямую. Воспроизведите некоторые файлы в каталоге *ESC-50*, чтобы услышать доступные звуки. Как только вы это сделаете, мы пойдем дальше.

Исследуя данные ESC-50

При работе с новым набором данных всегда полезно прочувствовать их *форму*, прежде чем погрузиться в построение моделей. Например, решая задачи классификации, вы должны узнать, действительно ли ваш набор данных содержит примеры из всех возможных классов, и в идеале нужно, чтобы все классы присутствовали в равных количествах. Давайте посмотрим, как работает ESC-50.

Мы знаем, что последний набор цифр в каждом имени файла описывает класс, к которому он принадлежит, поэтому нам нужно получить список файлов и подсчитать наличие каждого класса:

```
import glob
from collections import Counter

esc50_list = [f.split("-")[-1].replace(".wav","")
        for f in
        glob.glob("ESC-50/audio/*.wav")]
Counter(esc50_list)
```

Сначала мы формируем список имен файлов ESC-50. Поскольку нам важен только номер класса в конце имени файла, мы «отрезаем» расширение *.wav* и разделяем имя файла, выделяя сепаратор. Наконец, мы

берем последний элемент этой разделенной строки. Если вы проверите `esc50_list`, то получите пакет строк в диапазоне от 0 до 49. Мы могли бы написать больше кода, который компилирует `dict` и подсчитывает все экземпляры, но мне лень, поэтому я использую вспомогательную функцию Python `Counter`, которая делает все это за нас. Вот что получается:

```
Counter({'15': 40,
     '22': 40,
     '36': 40,
     '44': 40,
     '23': 40,
     '31': 40,
     '9': 40,
     '13': 40,
     '4': 40,
     '3': 40,
     '27': 40,
     …})
```

Это тот редкий случай, когда набор данных идеально сбалансирован. Давайте это отпразднуем и установим еще несколько библиотек, которые понадобятся в ближайшее время.

Если набор данных содержит *несбалансированный* объем данных, то вы можете просто случайным образом дублировать примеры меньших классов, пока не увеличите их до числа других классов. Хотя это похоже на фиктивный учет, на практике это удивительно эффективно (и дешево!).

SoX и LibROSA

Большая часть обработки аудиоданных, которую выполняет torchaudio, основана на двух других видах программного обеспечения: *SoX* и *LibROSA*. *LibROSA* — это библиотека Python для аудиоанализа, включающая генерацию мел-спектрограмм (вы узнаете, что это, чуть позже), обнаружение ритмов и даже создание музыки.

SoX — это программа, с которой вы, возможно, уже знакомы, если когда-либо пользовались Linux. На самом деле *SoX* даже старше, чем сам Linux; впервые она вышла в июле 1991 года, в то время как дебют Linux пришелся на сентябрь 1991 года. Я помню, как пользовался ею еще в 1997 году для

преобразования WAV-файлов в формат MP3 на своей первой Linux-box. Но эта программа до сих пор полезна![1]

Если вы устанавливаете `torchaudio` через `conda`, то можете перейти к следующему разделу. Если вы используете `pip`, вам, вероятно, потребуется установить саму *SoX*. Для системы на базе Red Hat введите следующее:

```
yum install sox
```

Для системы на базе Debian введите следующее:

```
apt intall sox
```

После установки *SoX* можете перейти непосредственно к `torchaudio`.

torchaudio

Установить `torchaudio` можно с помощью `conda` или `pip`:

```
conda install -c derickl torchaudio
pip install torchaudio
```

В отличие от torchvision, torchaudio похожа на torchtext тем, что ее тоже не так сильно любят, мало поддерживают и почти не обеспечивают документацией. Я надеюсь, что в ближайшем будущем это изменится, когда PyTorch станет популярнее и будут созданы более совершенные конвейеры обработки текста и аудио. Тем не менее для решения наших задач torchaudio будет вполне достаточно; нам просто нужно написать несколько специальных загрузчиков данных (чего не нужно было делать для обработки аудиоданных или текста).

В любом случае ядро torchaudio находится в `load()` и `save()`. В этой главе мы рассмотрим только `load()`, но вам нужно будет использовать `save()`, если вы создаете новый звук с вашего входа (например, модель преобразования текста в речь). `load()` принимает файл, указанный в `filepath`, и возвращает тензорное представление аудиофайла и частоту дискретизации этого аудиофайла в виде отдельной переменной. Теперь у нас есть средства

[1] Все функциональные возможности SoX (http://sox.sourceforge.net/) выходят за рамки этой книги и не нужны для решения наших задач в оставшейся части этой главы.

для загрузки одного из WAV-файлов из набора данных ESC-50 и преобразования его в тензор. В отличие от работы с текстом и изображениями, нам нужно написать чуть больше кода, прежде чем мы сможем приступить к созданию и обучению модели. Нам нужно написать собственный *набор данных*.

Создание набора данных ESC-50

Мы говорили о наборах данных в главе 2, но torchvision и torchtext сделали за нас всю тяжелую работу, поэтому нам не пришлось думать о деталях. Как вы, возможно, помните, в пользовательском наборе данных должны быть реализованы два метода класса `__getitem__` и `__len__`, чтобы загрузчик данных мог получить пакет тензоров и их маркировки, а также общее количество тензоров в наборе данных. У нас также есть метод `__init__` для настройки пути к файлам, которые будут неоднократно использоваться.

Вот наш первый проход в набор данных ESC-50:

```
class ESC50(Dataset):

    def __init__(self,path):
        #Get directory listing from path
        files = Path(path).glob('*.wav')
        #Iterate through the listing and create a list of tuples (
                              filename, label)
        self.items = [(f,int(f.name.split("-")[-1]
                    .replace(".wav",""))) for f in files]
        self.length = len(self.items)

    def __getitem__(self, index):
        filename, label = self.items[index]
        audio_tensor, sample_rate = torchaudio.load(filename)
        return audio_tensor, label

    def __len__(self):
        return self.length
```

Большая часть работы в классе происходит, когда создается его новый экземпляр.

Метод `__init__` принимает параметр `path`, находит все WAV-файлы внутри этого пути и затем создает кортежи (filename, label), используя тот же разделитель строк, о котором я уже писал в этой главе, чтобы получить маркировку этого аудиосемпла. Когда PyTorch запрашивает элемент из на-

бора данных, мы индексируем его в список `items`, используем `torchaudio.load`, чтобы заставить `torchaudio` загрузить аудиофайл, превращаем его в тензор, а затем возвращаем как тензор, так и маркировку.

Для начала этого достаточно. Для проверки работоспособности давайте создадим объект `ESC50` и извлечем первый элемент:

```
test_esc50 = ESC50(PATH_TO_ESC50)
tensor, label = list(test_esc50)[0]

tensor
tensor([-0.0128, -0.0131, -0.0143,...,0.0000,0.0000,0.0000])

tensor.shape
torch.Size([220500])

label
'15'
```

Мы можем создать загрузчик данных, используя стандартные структуры PyTorch:

```
example_loader = torch.utils.data.DataLoader(test_esc50, batch_size = 64,
shuffle = True)
```

Но прежде чем сделать это, следует вернуться к нашим данным. Как вы помните, мы всегда должны создавать обучение, валидацию и набор данных для тестов. На данный момент у нас есть только один каталог со всеми данными, что не подходит. Разделение данных в пропорции 60/20/20 на обучающие, валидационные и тестовые соответствует нашим целям. Итак, мы можем это сделать, взяв случайную выборку из всего набора данных (обеспечив выборку без замены и убедившись, что только что созданные наборы данных сбалансированы), но опять же, набор данных ESC-50 избавляет нас от необходимости выполнять большую работу. Компиляторы набора данных разделили данные на пять равных сбалансированных *фолдов*, обозначенных *первой* цифрой в имени файла. У нас будут фолды `1,2,3` — набор данных для обучения, `4` — набор данных для валидации и `5` — набор данных для теста. Но если вы не хотите быть скучным и последовательным, не бойтесь их смешивать! Переместите каждый фолд в каталоги *тестирования, обучения и валидации*:

```
mv 1* ../train
mv 2* ../train
```

```
mv 3* ../train
mv 4* ../valid
mv 5* ../test
```

Теперь мы можем создавать отдельные наборы данных и загрузчики:

```
from pathlib import Path

bs=64
PATH_TO_ESC50 = Path.cwd() / 'esc50'
path ='test.md'
test

train_esc50 = ESC50(PATH_TO_ESC50 / "train")
valid_esc50 = ESC50(PATH_TO_ESC50 / "valid")
test_esc50= ESC50(PATH_TO_ESC50 / "test"

train_loader = torch.utils.data.DataLoader(train_esc50, batch_size = bs,
               shuffle = True)
valid_loader = torch.utils.data.DataLoader(valid_esc50, batch_size = bs,
               shuffle = True)
test_loader  = torch.utils.data.DataLoader(test_esc50, batch_size = bs,
               shuffle = True)
```

Все данные готовы, поэтому мы можем посмотреть на модель классификации.

Модель CNN для набора данных ESC-50

Для первой попытки классификации звуков мы создаем модель, которая в значительной степени построена на базе статьи под названием «Very Deep Convolutional Networks For Raw Waveforms».[1] Вы увидите, что в ней используется множество составляющих, о которых я писал в главе 3, но вместо 2D-слоев мы используем 1D-слои, поскольку в нашем случае мы имеем на одну размерность меньше:

```
class AudioNet(nn.Module):
    def __init__(self):
        super(AudioNet, self).__init__()
        self.conv1 = nn.Conv1d(1, 128, 80, 4)
        self.bn1 = nn.BatchNorm1d(128)
        self.pool1 = nn.MaxPool1d(4)
```

[1] См. «Very Deep CNN for Raw Waveforms», Wei Dai et al. (2016) (https://arxiv.org/pdf/1610.00087.pdf).

```
        self.conv2 = nn.Conv1d(128, 128, 3)
        self.bn2 = nn.BatchNorm1d(128)
        self.pool2 = nn.MaxPool1d(4)
        self.conv3 = nn.Conv1d(128, 256, 3)
        self.bn3 = nn.BatchNorm1d(256)
        self.pool3 = nn.MaxPool1d(4)
        self.conv4 = nn.Conv1d(256, 512, 3)
        self.bn4 = nn.BatchNorm1d(512)
        self.pool4 = nn.MaxPool1d(4)
        self.avgPool = nn.AvgPool1d(30)
        self.fc1 = nn.Linear(512, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(self.bn1(x))
        x = self.pool1(x)
        x = self.conv2(x)
        x = F.relu(self.bn2(x))
        x = self.pool2(x)
        x = self.conv3(x)
        x = F.relu(self.bn3(x))
        x = self.pool3(x)
        x = self.conv4(x)
        x = F.relu(self.bn4(x))
        x = self.pool4(x)
        x = self.avgPool(x)
        x = x.permute(0, 2, 1)
        x = self.fc1(x)
        return F.log_softmax(x, dim = 2)
```

Также нужен оптимизатор и функция потерь. В качестве оптимизатора, как и прежде, мы используем Adam, но как вы думаете, какую функцию потерь следует использовать? (Если вы ответили `CrossEn tropyLoss`, то вам полагается медалька!)

```
audio_net = AudioNet()
audio_net.to(device)
```

Создав нашу модель, мы сохраняем веса и используем функцию `find_lr()` из главы 4:

```
audio_net.save("audionet.pth")
import torch.optim as optim
optimizer = optim.Adam(audionet.parameters(), lr=0.001)
logs,losses = find_lr(audio_net, nn.CrossEntropyLoss(), optimizer)
plt.plot(logs,losses)
```

По графику на рис. 6.3 определяем, что соответствующая скорость обучения составляет около `1e-5` (в зависимости от того, где спуск выглядит наиболее крутым). Задаем это значение как скорость обучения и перезагружаем начальные веса нашей модели:

```
lr = 1e-5
model.load("audionet.pth")
import torch.optim as optim
optimizer = optim.Adam(audionet.parameters(), lr=lr)
```

Обучаем модель 20 эпох:

```
train(audio_net, optimizer, torch.nn.CrossEntropyLoss(),
train_data_loader, valid_data_loader, epochs=20)
```

После обучения вы обнаружите, что модель выдает примерно 13–17 % точности на нашем наборе данных. Это лучше, чем 2 %, на которые мы могли бы рассчитывать, если бы просто рандомно выбирали один из 50 классов. Но, вероятно, результат можно улучшить. Давайте рассмотрим другой способ просмотра наших аудиоданных.

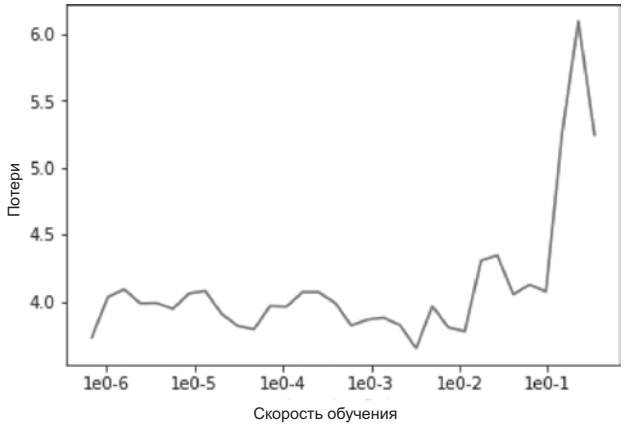

Рис. 6.3. График скорости обучения AudioNe

Частота — моя вселенная

Если вы прочитаете о ESC-50 на странице GitHub, то увидите список лидеров сетевых архитектур и их коэффициенты точности.

Вы заметите, что по сравнению с ними у нас не все так радужно. Можно было бы расширить созданную модель, чтобы сделать ее глубже и тем самым немного повысить точность, но для реального увеличения производительности нужно менять домены. При обработке аудиоданных вы можете работать с чистыми формами волн, что мы и делали; но большую часть времени придется работать в диапазоне частот. Это различное представление превращает необработанную форму волны в представление данных, которое показывает все частоты звука в данный момент времени. Возможно, это более информационно насыщенное представление нейронной сети, поскольку оно сможет работать с этими частотами напрямую, и нам не нужно выяснять, как преобразовать необработанный сигнал в ту форму, которую модель сможет использовать.

Давайте посмотрим, как генерировать частотные спектрограммы с помощью LibROSA.

Мел-спектрограммы

Традиционно, чтобы попасть в диапазон частот, к звуковому сигналу необходимо применить преобразование Фурье. Мы пойдем немного дальше, создав спектрограммы в мел-шкале. *Мел-шкала* определяет шкалу высот звука, равноудаленных друг от друга, где 1000 мел = 1000 Гц. Эта шкала обычно используется при обработке аудиоданных, особенно для распознавания и классификации речи. Для создания мел-спектрограммы с помощью LibROSA можно обойтись двумя строчками кода:

```
sample_data, sr = librosa.load("ESC-50/train/1-100032-A-0.wav", sr=None)
spectrogram = librosa.feature.melspectrogram(sample_data, sr=sr)
```

В результате получаем массив NumPy, содержащий данные спектрограммы. Если мы отобразим эту спектрограмму, как показано на рис. 6.4, то увидим частоты в нашем звуке:

```
librosa.display.specshow(spectrogram, sr=sr, x_axis='time', y_axis='mel')
```

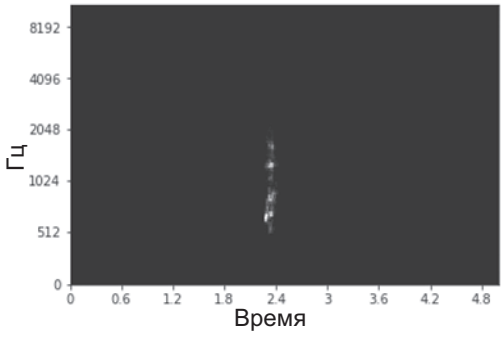

Рис. 6.4. Мел-спектрограмма

На изображении не так много информации. Можно это улучшить! Если конвертировать спектрограмму в логарифмическую шкалу, то структуры звука станут лучше, поскольку шкала будет представлять более широкий диапазон значений. В обработке аудиоданных этот метод достаточно распространен, и у LibROSA есть для этого специальный код:

```
log_spectrogram = librosa.power_to_db(spectrogram, ref=np.max)
```

Таким образом вычисляется коэффициент масштабирования `10 * log10(spectrogram / ref)`. `ref` по умолчанию равно `1.0`, но здесь мы передаем `np.max()`, так что `spectrogram / ref` попадет в диапазон `[0,1]`. На рис. 6.5 показана новая спектрограмма.

Теперь у нас есть логарифмическая мел-спектрограмма! Если вы вызовете `log_spectrogram.shape`, то увидите, что это 2D-тензор, что логично, так как мы построили график изображений с помощью тензора. Мы могли бы создать новую архитектуру нейронной сети и передать в нее новые данные, но у меня в рукаве припасен дьявольский трюк. Буквально только что мы сгенерировали изображения данных спектрограммы. Почему бы не поработать с ними?

Поначалу такое решение может показаться глупым; в конце концов, у нас есть основные данные спектрограммы, которые точнее, чем представление изображения (если мы видим, что значение точки данных соответствует 58, а не 60, то для нас это *значит* больше, чем другой оттенок, например фиолетовый). И если бы мы начинали с нуля, так бы оно и было. Но! У нас уже есть обученные сети, такие как ResNet и Inception, которые, как

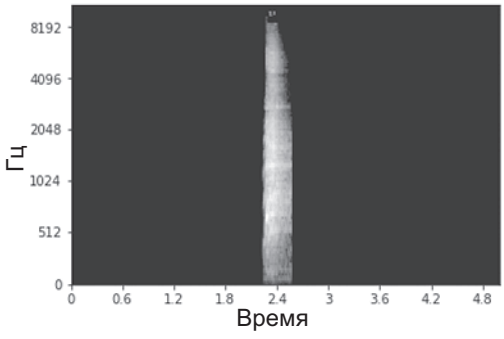

Рис. 6.5. Логарифмическая мел-спектрограмма

мы знаем, прекрасно распознают структуру и другие части изображений. Мы можем создавать представления изображений нашего аудио и использовать предварительно обученную сеть, чтобы значительно улучшить точность при небольшом обучении, используя сверхмощный перенос обучения еще раз. Для нашего набора данных это может быть подходящим решением, так как у нас не так много примеров для обучения сети (всего 2000!).

Этот прием можно использовать во многих разнородных наборах данных. Если вы найдете способ, как можно недорого преобразовать данные в представление изображения, стоит им воспользоваться и применить сеть ResNet, чтобы оценить, на что способен перенос обучения, и иметь представление о том, с чем придется столкнуться, если вы решите воспользоваться другим подходом. Теперь давайте создадим новый набор данных, который будет генерировать эти изображения по запросу.

Новый набор данных

Отбросьте исходный класс набора данных `ESC50` и создайте новый `ESC50Spectro gram`. Несмотря на то что это отделит некоторую часть кода от старого класса, в этой версии методом `__get_item__` можно получить гораздо больше. Мы генерируем спектрограмму с помощью *LibROSA*, а затем делаем некоторые манипуляции с `matplotlib`, чтобы получить данные в массив NumPy. Мы применяем массив к нашему конвейеру преобразования (который просто использует `ToTensor`) и возвращаем его и маркировку элемента. Вот код, который мы для этого используем:

```
class ESC50Spectrogram(Dataset):

def __init__(self,path):
    files = Path(path).glob('*.wav')
    self.items = [(f,int(f.name.split("-")[-1].replace(".wav","")))
                  for f in files]
    self.length = len(self.items)
    self.transforms = torchvision.transforms.Compose(
                  [torchvision.transforms.ToTensor()])

def __getitem__(self, index):
    filename, label = self.items[index]
    audio_tensor, sample_rate = librosa.load(filename, sr=None)
    spectrogram = librosa.feature.melspectrogram(audio_tensor, sr=sample_rate)
    log_spectrogram = librosa.power_to_db(spectrogram, ref=np.max)
    librosa.display.specshow(log_spectrogram, sr=sample_rate,
                             x_axis='time', y_axis='mel')
    plt.gcf().canvas.draw()
    audio_data = np.frombuffer(fig.canvas.tostring_rgb(), dtype=np.uint8)
    audio_data = audio_data.reshape(fig.canvas.get_width_height()
                             [::-1] + (3,))
    return (self.transforms(audio_data), label)

def __len__(self):
    return self.length
```

Мы не станем тратить слишком много времени на эту версию набора данных, потому что в ней есть большой недостаток, который я продемонстрирую с помощью метода Python `process_time()`:

```
oldESC50 = ESC50("ESC-50/train/")
start_time = time.process_time()
oldESC50.__getitem__(33)
end_time = time.process_time()
old_time = end_time - start_time

newESC50 = ESC50Spectrogram("ESC-50/train/")
start_time = time.process_time()
newESC50.__getitem__(33)
end_time = time.process_time()
new_time = end_time - start_time

old_time = 0.004786839000075815
new_time = 0.39544327499993415
```

Новый набор данных почти в сто раз медленнее, чем исходный, который только что вернул необработанный звук! Обучение становится невероятно медленным и может даже свести на нет любые преимущества переноса обучения.

Мы можем использовать несколько способов, которые решают большинство наших проблем. Первый подход заключается в добавлении кэша для хранения сгенерированной спектрограммы в памяти, поэтому нам не нужно восстанавливать ее каждый раз, когда мы вызываем метод `__getitem__`. С пакетом Python `functools` это легко:

```
import functools

class ESC50Spectrogram(Dataset):
 #skipping init code

    @functools.lru_cache(maxsize=<size of dataset>)
    def __getitem__(self, index):
```

При условии, что оперативной памяти достаточно для хранения всего содержимого набора данных, этого может быть вполне достаточно. Мы настроили *давно не используемый кэш* (LRU), который будет сохранять содержимое в памяти как можно дольше, причем при загрузке памяти индексы, к которым давно не обращались, будут извлекаться из кэша в первую очередь. Однако если у вас недостаточно памяти для хранения, на каждой итерации пакета вы столкнетесь с медленной работой, так как удаленные спектрограммы придется восстанавливать. Я предпочитаю *предварительно вычислить* все возможные графики, а затем создать новый пользовательский класс набора данных, который загружает эти изображения с диска. (Вы также можете добавить аннотацию кэша LRU для дальнейшего ускорения.)

Для *предварительного вычисления* не нужно делать ничего сверхъестественного, это просто метод, который сохраняет графики в том же каталоге, через который они проходят:

```
def precompute_spectrograms(path, dpi=50):
    files = Path(path).glob('*.wav')
    for filename in files:
        audio_tensor, sample_rate = librosa.load(filename, sr=None)
        spectrogram = librosa.feature.melspectrogram(audio_tensor, sr=sr)
        log_spectrogram = librosa.power_to_db(spectrogram, ref=np.max)
        librosa.display.specshow(log_spectrogram, sr=sr, x_axis='time',
                                 y_axis='mel')
        plt.gcf().savefig("{}{}_{}.png".format(filename.parent, dpi,
                          filename.name),dpi=dpi)
```

Этот метод проще, чем наш предыдущий набор данных, потому что мы можем использовать метод `savefig` от matplotlib для сохранения графика

непосредственно на диск, вместо того чтобы возиться с NumPy. Мы также предоставляем дополнительный входной параметр, `dpi`, который позволяет нам контролировать качество сгенерированных выходных данных. Запустите его на всех путях обучения, тестирования и валидации, которые мы уже настроили (скорее всего, потребуется несколько часов, чтобы пройти через все изображения).

Все, что сейчас нужно, — это новый набор данных, который читает эти изображения. Мы не можем использовать стандартный `ImageDataLoader`, о котором шла речь в главах 2–4, поскольку схема имени файла PNG не соответствует используемой им структуре каталогов. Но независимо от этого, мы можем просто открыть изображение с помощью Python Imaging Library:

```
from PIL import Image

    class PrecomputedESC50(Dataset):
        def__init__(self,path,dpi=50, transforms=None):
           files = Path(path).glob('{}*.wav.png'.format(dpi))
           self.items = [(f,int(f.name.split("-")[-1]
           .replace(".wav.png",""))) for f in files]
           self.length = len(self.items)
           if transforms=None:
               self.transforms =
               torchvision.transforms.Compose([torchvision.transforms.
                                               ToTensor()])
           else:
               self.transforms = transforms

        def __getitem__(self, index):
            filename, label = self.items[index]
            img = Image.open(filename)
            return (self.transforms(img), label)

        def __len__(self):
            return self.length
```

Этот код намного проще, и, надеюсь, это видно даже по времени, которое требуется для получения элемента из набора данных:

```
start_time = time.process_time()
b.__getitem__(33)
end_time = time.process_time()
end_time - start_time
>> 0.0031465259999094997
```

Получение элемента из этого набора данных занимает примерно то же время, что и в нашем исходном подходе к работе с аудиофайлами, поэтому мы ничего не теряем, перейдя к подходу, основанному на анализе изображений, за исключением того, что нам придется поначалу предварительно обработать все изображения, прежде чем мы создадим базу данных. Мы также добавили конвейер преобразования по умолчанию, который преобразует изображение в тензор, но во время инициализации его можно заменить на другой конвейер. Вооружившись этими опциями, можем начать перенос обучения.

Появление ResNet

Как вы, наверное, помните из главы 4, перенос обучения требует, чтобы мы взяли модель, которая уже была обучена для определенного набора данных (в случае с изображениями это, вероятно, ImageNet), и настроили ее на конкретной предметной области, наборе данных ESC-50, который преобразуем в спектрограммы. Возможно, вы хотите знать, полезна ли модель, обученная на *обычных* фотографиях.

Оказывается, что предварительно обученные модели *обучаются* многим структурам, применимым к доменам, которые на первый взгляд могут показаться абсолютно разными. Вот код из главы 4, который инициализирует модель:

```
from torchvision import models
spec_resnet = models.ResNet50(pretrained=True)

for param in spec_resnet.parameters():
    param.requires_grad = False

spec_resnet.fc = nn.Sequential(nn.Linear(spec_resnet.fc.in_features,500),
nn.ReLU(),
nn.Dropout(), nn.Linear(500,50))
```

Он инициализирует предварительно обученную (и замороженную) модель `ResNet50` и заменяет «голову» модели на необученный модуль `Sequential`, который заканчивается `Linear` с выходом 50, по одному для каждого из классов в наборе данных ESC-50. Также необходимо создать `DataLoader`, который будет принимать предварительно вычисленные спектрограммы. Когда мы создаем набор данных ESC-50, также нужно нормализовать

входящие изображения со стандартным и средним отклонением ImageNet, так как именно на них обучалась предварительно обученная архитектура ResNet-50. Можно сделать это, передав новый конвейер:

```
esc50pre_train = PreparedESC50(PATH, transforms=torchvision.transforms
.Compose([torchvision.transforms.ToTensor(),
torchvision.transforms.Normalize
(mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])]))

esc50pre_valid = PreparedESC50(PATH, transforms=torchvision.transforms
.Compose([torchvision.transforms.ToTensor(),
torchvision.transforms.Normalize
(mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])]))

esc50_train_loader = (esc50pre_train, bs, shuffle=True)
esc50_valid_loader = (esc50pre_valid, bs, shuffle=True)
```

Установив загрузчики данных, теперь можно перейти к определению скорости обучения и начать к нему готовиться.

Определение скорости обучения

Требуется определить скорость обучения для ее использования в модели. Как и в главе 4, сохраним исходные параметры модели и используем функцию `find_lr()`, чтобы найти подходящую скорость обучения. На рис. 6.6 показан график потерь по отношению к скорости обучения.

```
spec_resnet.save("spec_resnet.pth")
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(spec_resnet.parameters(), lr=lr)
logs,losses = find_lr(spec_resnet, loss_fn, optimizer)
plt.plot(logs, losses)
```

Глядя на график скорости обучения, построенный по отношению к потерям, кажется, что значение `1e-2` вполне подходит. Поскольку модель ResNet-50 несколько глубже, чем предыдущая, мы возьмем дифференциальную скорость обучения `[1e-2,1e-4,1e-8]` с самой высокой скоростью обучения, примененной к нашему классификатору (так как это требует больше обучения!), и меньшей скоростью для уже обученного канала бэкбон. Опять же, используя Adam в качестве оптимизатора, не пренебрегайте другими доступными вариантами.

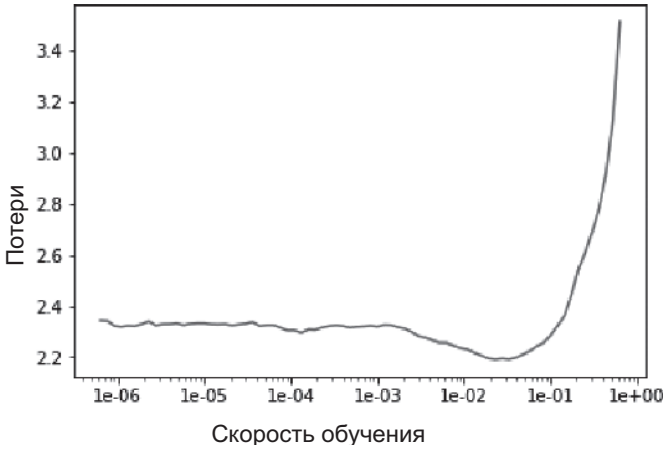

Рис. 6.6. График скорости обучения SpecResNet

Однако прежде чем применять эти дифференциальные скорости, мы обучаемся в течение нескольких эпох, которые обновляют только классификатор, поскольку мы *заморозили* бэкбон ResNet-50 при создании нашей сети:

```
optimizer = optim.Adam(spec_resnet.parameters(), lr=[1e-2,1e-4,1e-8])

train(spec_resnet, optimizer, nn.CrossEntropyLoss(),
esc50_train_loader, esc50_val_loader,epochs=5,device="cuda")
```

Теперь мы размораживаем бэкбон и применяем дифференциальные скорости:

```
for param in spec_resnet.parameters():
    param.requires_grad = True

optimizer = optim.Adam(spec_resnet.parameters(), lr=[1e-2,1e-4,1e-8])

train(spec_resnet, optimizer, nn.CrossEntropyLoss(),
esc50_train_loader, esc50_val_loader,epochs=20,device="cuda")

> Epoch 19, accuracy = 0.80
```

Как видите, имея точность валидации около 80 %, мы уже значительно опережаем нашу исходную модель `AudioNet`. Возможности переноса обучения снова побеждают! Не бойтесь обучать сеть за большее количе-

ство эпох, чтобы увидеть, улучшается ли точность. Если мы посмотрим на рейтинговую таблицу ESC-50, то увидим точность, равную уровню человека. И это только ResNet-50. Чтобы поднять оценку еще выше, можете попробовать ResNet-101 и, возможно, даже ансамбль различных архитектур.

Перейдем к аугментации данных. Давайте рассмотрим несколько способов сделать это в обоих доменах, над которыми мы до сих пор работали.

Аугментация аудиоданных

Из главы 4, посвященной работе с изображениями, мы узнали, что повысить точность классификатора можно путем внесения изменений во входящие изображения. Отражая их, обрезая или применяя другие преобразования, мы усложнили работу нейронной сети на этапе обучения и в итоге получили более *генерализированную* модель, которая просто не соответствовала представленным данным (не забывайте о проблеме переобучения). Можем ли мы сделать точно так же? Да! Фактически мы можем использовать два подхода — один из них очевидный и работает с исходной формой звуковой волны, а второй, возможно, менее очевидный и следует из нашего решения использовать классификатор на основе ResNet для изображений мел-спектрограммы. Давайте для начала разберем аудиопреобразования.

Преобразования torchaudio

Как и torchvision, torchaudio включает в себя модуль `transforms`, который выполняет преобразования входящих данных. Тем не менее количество предлагаемых преобразований невелико, особенно по сравнению с количеством, которое мы получаем, когда работаем с изображениями. Если вам интересно, то полный список можно посмотреть в документации. В этой книге мы остановимся только на одном из преобразований — это `torchaudio.transforms.PadTrim`. В случае с набором данных ESC-50 нам повезло, так как каждый аудиоклип имеет одинаковую длину. В реальном мире это, конечно, не встречается, но наши нейронные сети любят (и иногда настаивают на этом, в зависимости от того, как они построены), чтобы входные данные были примерно одного размера. `PadTrim` возьмет входящий звуковой тензор и либо доведет его до не-

обходимой длины, либо обрежет, чтобы он не превышал эту длину. Если нам нужно обрезать клип, чтобы получить другую длину, используйте `PadTrim` следующим образом:

```
audio_tensor, rate = torchaudio.load("test.wav")
audio_tensor.shape
trimmed_tensor = torchaudio.transforms.PadTrim(max_len=1000)(audio_orig)
```

Однако если вы хотите получить аугментацию, которая фактически изменяет звучание звука (например, добавляет эхо, шум или изменяет темп клипа), то вам не нужен модуль `torchaudio.transforms`. Используйте вместо него *SoX*.

Эффекты SoX

Не знаю почему, но он не является частью модуля `transforms`, однако `torchaudio.sox_effects.SoxEffectsChain` позволяет создавать цепочку из одного или нескольких SoX-эффектов и применять их ко входному файлу. Здесь не самый удобный интерфейс, поэтому давайте посмотрим, как он работает в новой версии набора данных, изменяющей высоту звукового файла:

```
class ESC50WithPitchChange(Dataset):
    def __init__(self,path):
        #Получить список каталогов из файлов пути = Path(path).glob('*.wav')
        #Просмотрите список и создайте список кортежей
                 (filename, label)
        self.items = [(f,f.name.split("-")[-1].replace(".wav",""))
                      for f in files]
        self.length = len(self.items)
        self.E = torchaudio.sox_effects.SoxEffectsChain()
        self.E.append_effect_to_chain("pitch", [0.5])

    def __getitem__(self, index):
        filename, label = self.items[index]
        self.E.set_input_file(filename)
        audio_tensor, sample_rate = self.E.sox_build_flow_effects()
        return audio_tensor, label

    def __len__(self):
        return self.length
```

В методе `__init__` мы создаем новую переменную экземпляра `E`, `SoxEffectsChain`, которая будет содержать все эффекты, которые мы хотим применить к нашим аудиоданным. Затем с помощью `append_effect_to_`

chain мы добавляем новый эффект, который принимает строку, указывающую имя эффекта, и массив параметров для отправки в sox. Вы можете получить список доступных эффектов, вызвав torchaudio.sox_effects.effect_names(). Если бы нужно было добавить еще один эффект, это произошло бы после того, как мы настроили эффект основной высоты, поэтому если вы хотите создать список отдельных эффектов и применить их случайным образом, вам нужно будет создать отдельные цепочки для каждого из них. Когда дело доходит до выбора элемента для возврата в загрузчик данных, все работает немного по-другому. Вместо torchaudio.load() мы обращаемся к нашей цепочке эффектов и направляем ее в файл с помощью set_input_file. Обратите внимание, что эта команда не загружает файл! Для этого мы должны использовать sox_build_flow_effects(), который запускает *SoX* в фоновом режиме, применяет эффекты в цепочке и возвращает информацию о тензорне и частоте дискретизации, которую мы в противном случае получили бы из load().

SoX способен на довольно многое, и я не буду подробно рассказывать обо всех возможных эффектах, которые вы можете получить. Предлагаю ознакомиться с документацией SoX (https://oreil.ly/uLBTF), а также изучить list_effects() и все его возможности.

Такие преобразования позволяют нам изменять исходное аудио, но в этой главе немалое время отведено созданию конвейера обработки, который работает с изображениями мел-спектрограмм. Мы могли бы сделать то же самое, что делали для генерации исходного набора данных для этого конвейера, создав измененные аудиосемплы, а из них спектрограммы, но на этой стадии мы генерируем очень много данных, которые нам нужно будет смешивать во время прогона. К счастью, мы можем осуществить некоторые преобразования на самих спектрограммах.

SpecAugment

Теперь вы можете подумать: «Подождите, эти спектрограммы — просто изображения! Мы можем использовать любое преобразование изображения, какое захотим!» И да! В яблочко. Но будьте осторожны; случайный обрез может вырезать достаточное количество частот, чтобы потенциально изменить класс выходных данных.

Нашего набора данных ESC-50 это касается в гораздо меньшей степени, но если вы делаете что-то вроде распознавания речи, при аугментации

вам будет необходимо это учитывать. Другая интересная возможность заключается в том, что поскольку мы знаем, что все спектрограммы имеют одинаковую структуру (они всегда будут представлять собой частотный график!), мы могли бы создавать преобразования, основанные на анализе изображений, которые работают конкретно вокруг этой структуры.

В 2019 году Google выпустила статью о SpecAugment,[1] в которой говорилось о новых современных результатах по многим наборам аудиоданных. Команда получила эти результаты, используя три новых метода аугментации данных, которые они применили непосредственно к мел-спектрограмме: алгоритм динамической трансформации временной шкалы, частотное маскирование и временное маскирование. Мы не будем рассматривать алгоритм динамической трансформации временной шкалы, потому что нам от этого нет никакой пользы, однако мы будем применять специальные преобразования для частотного и временного маскирования.

Частотное маскирование

Частотное маскирование случайным образом удаляет частоту или набор частот из аудио входных данных. Это делает работу модели тяжелее; она просто не может запомнить входные данные и их класс, потому что у них будут разные частоты, маскируемые в каждом пакете. Вместо этого модели придется обучаться другим функциям, которые могут определить, как сопоставить входные данные с классом, что должно в результате дать более точную модель. На нашей мел-спектрограмме это подтверждается тем, что спектрограф ничего не отображает для этой частоты в любой момент времени. Рисунок 6.7 показывает, как это выглядит: по сути, пустая линия, проведенная через естественную спектрограмму.

Так выглядит код для пользовательского `Transform`, который реализует частотное маскирование:

```
class FrequencyMask(object):
    """
      Пример:
        >>> transforms.Compose([
        >>>     transforms.ToTensor(),
        >>>     FrequencyMask(max_width=10, use_mean=False),
        >>> ])
```

[1] См. «SpecAugment: A Simple Data Augmentation Method for Automatic Speech Recognitio», Daniel S. Park et al. (2019) (https://arxiv.org/abs/1904.08779).

```
    """

    def__init__(self, max_width, use_mean=True):
        self.max_width = max_width
        self.use_mean = use_mean

    def __call__(self, tensor):
        """
        Args:
            tensor (Tensor): Тензорное изображение размера (C, H, W),
            где должна применяться частотная маска.

        Returns:
            Tensor: Преобразованное изображение с частотной маской.
        """
        start = random.randrange(0, tensor.shape[2])
        end = start + random.randrange(1, self.max_width)
        if self.use_mean:
            tensor[:, start:end, :] = tensor.mean()
        else:
            tensor[:, start:end, :] = 0
        return tensor

    def __repr__(self):
        format_string = self.__class__.__name__ + "(max_width="
        format_string += str(self.max_width) + ")"
        format_string += 'use_mean=' + (str(self.use_mean) + ')')

        return format_string
```

После применения преобразования PyTorch вызывает метод `__call__` с тензорным представлением изображения (поэтому нам нужно поместить его в цепочку `Compose` после преобразования изображения в тензор, а не до). Мы предполагаем, что тензор будет иметь формат *каналы* × *высота* × *ширина*, и нам нужно задать значения высоты в небольшом диапазоне, равном нулю или среднему значению изображения (поскольку мы используем логарифмические спектрограммы, среднее значение должно быть равно нулю, но мы включаем оба варианта, чтобы вы могли поэкспериментировать и понять, что лучше).

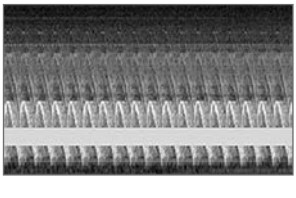

Рис. 6.7. Частотная маска, примененная к случайной выборке ESC-50

Диапазон обеспечивается параметром `max_width`, и наша полученная в результате пиксельная маска будет иметь ширину от 1 до `max_pixels`. Нам также нужно выбрать произвольную начальную точку маски, для чего используется переменная `start`. Наконец, мы подошли к сложной части этого преобразования — мы применяем нашу сгенерированную маску:

```
tensor[:, start:end, :] = tensor.mean()
```

Все не так уж и плохо. Наш тензор имеет три размерности, но нужно применить это преобразование ко всем красным, зеленым и синим каналам, поэтому используем пустой оператор: чтобы выбрать все в этой размерности. Используя `start:end`, мы выбираем диапазон высоты, а затем все, что есть в канале ширины, так как нам нужно применить маску на каждом временном шаге. А затем в правой части выражения задаем значение; в этом случае `tensor.mean()`. Если мы возьмем случайный тензор из набора данных ESC-50 и применим к нему преобразование, то, как видно на рис. 6.7, этот класс создает требуемую маску.

```
torchvision.transforms.Compose([FrequencyMask(max_width=10, use_mean=False),
torchvision.transforms.ToPILImage()])(torch.rand(3,250,200))
```

Далее обратим внимание на временное маскирование.

Временное маскирование

Закончив частотную маску, теперь мы можем перейти к *временной маске*, которая делает то же самое, что и частотная, но во временной области. Код здесь в основном такой же:

```
class TimeMask(object):
    """
      Example:
        >>> transforms.Compose([
        >>>     transforms.ToTensor(),
        >>>     TimeMask(max_width=10, use_mean=False),
        >>> ])
    """

    def __init__(self, max_width, use_mean=True):
       self.max_width = max_width
       self.use_mean = use_mean

def __call__(self, tensor):
    """
```

```
    Args:
        tensor (Tensor): тензорное изображение размера (C, H, W),
        в котором должно применяться временное маскирование.

    Returns:
        Tensor: преобразованное изображение с Time Mask.
    """
    start = random.randrange(0, tensor.shape[1])
    end = start + random.randrange(0, self.max_width)
    if self.use_mean:
        tensor[:, :, start:end] = tensor.mean()
    else:
        tensor[:, :, start:end] = 0
    return tensor

def __repr__(self):
    format_string = self.__class__.__name__ + "(max_width="
    format_string += str(self.max_width) + ")"
    format_string += 'use_mean=' + (str(self.use_mean) + ')')
    return format_string
```

Как видите, этот класс похож на частотную маску. Единственное отличие состоит в том, что наша переменная `start` теперь располагается в точке на оси высоты, и когда мы делаем маскирование, это выглядит следующим образом:

```
tensor[:, :, start:end] = 0
```

Это указывает на то, что мы выбираем все значения первых двух размерностей тензора и диапазона `start:end` в последней размерности. И снова мы можем применить это к случайному тензору из базы данных ESC-50, чтобы увидеть, что маска применяется правильно, как показано на рис. 6.8.

```
torchvision.transforms.Compose([TimeMask(max_width=10, use_mean=False),
torchvision.transforms.ToPILImage()])(torch.rand(3,250,200))
```

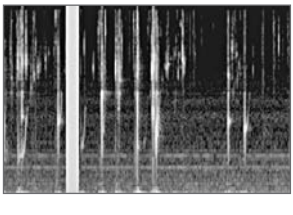

Рис. 6.8. Временная маска, примененная к случайной выборке ESC-50

Для завершения аугментации мы создаем новое оберточное преобразование, которое обеспечивает применение одной или обеих масок к изображению спектрограммы:

```
class PrecomputedTransformESC50(Dataset):
    def __init__(self,path,dpi=50):
        files = Path(path).glob('{}*.wav.png'.format(dpi))
        self.items = [(f,f.name.split("-")[-1].replace(".wav.png",""))
                      for f in files]
        self.length = len(self.items)
        self.transforms = transforms.Compose([
    transforms.ToTensor(),
    RandomApply([FrequencyMask(self.max_freqmask_width)]p=0.5),
    RandomApply([TimeMask(self.max_timemask_width)]p=0.5)
])

    def __getitem__(self, index):
        filename, label = self.items[index]
        img = Image.open(filename)
        return (self.transforms(img), label)

    def __len__(self):
        return self.length
```

Попробуйте повторить цикл обучения с этой аугментацией данных и посмотрите, сможете ли вы, как и Google, добиться большей точности с помощью этих масок. Но может быть, мы можем попробовать кое-что еще?

Новые эксперименты

Итак, мы создали две нейронные сети — одну на основе необработанной формы звуковой волны, а другую на основе изображений мел-спектрограммы для классификации звуков из набора данных ESC-50. Хотя вы уже убедились, что модель на основе ResNet является более точной и использует возможности переноса обучения, было бы интересно поэкспериментировать и создать комбинацию из двух сетей, чтобы понять, увеличивает она точность или, наоборот, уменьшает. Простой способ сделать это — вернуться к ансамблированию из главы 4: просто объединить и усреднить прогнозы. Кроме того, мы не описали идею построения сети на основе необработанных данных спектрограмм. Если создается модель,

которая работает с этими данными, помогает ли это общей точности, если она вводится в ансамбль? Мы также можем использовать другие версии ResNet или создать новую архитектуру, которая использует различные предварительно обученные модели, такие как VGG или Inception. Изучите некоторые из этих вариантов и попробуйте применить их на практике; мои эксперименты показывают, что SpecAugment улучшает точность классификации ESC-50 примерно на 2 %.

Заключение

В этой главе мы использовали две очень разные стратегии классификации аудио, кратко ознакомились с библиотекой torchaudio PyTorch и научились предварительно вычислять преобразования на наборах данных на случай, если мгновенное выполнение преобразований серьезно повлияет на время обучения. Мы поговорили о двух подходах к аугментации данных. В качестве неожиданного бонуса мы снова вернулись к обучению модели на основе анализа изображений, используя перенос обучения для быстрой генерации классификатора с приличной точностью в сравнении с другими классификаторами, представленными в таблице лидеров базы данных ESC-50.

На этом мы завершаем наш экскурс в область изображений, тестов и аудиофайлов, хотя мы к ним вернемся в главе 9, когда будем рассматривать другие задачи, для которых используется PyTorch. Далее, однако, мы разберемся, как отлаживать модели, если они не обучаются так, как нужно, или делают это недостаточно быстро.

Дополнительные источники

- «Interpreting and Explaining Deep Neural Networks for Classification of Audio Signals», Sören Becker et al. (2018), https://arxiv.org/abs/1807.03418
- «CNN Architectures for Large-Scale Audio Classification», Shawn Hershey et al. (2016), https://arxiv.org/abs/1609.09430v2

ГЛАВА 7

Отладка моделей PyTorch

В предыдущих главах мы уже создали много моделей, а теперь кратко рассмотрим их интерпретацию и выясним, что происходит за кулисами. Рассмотрим использование карты активаций класса (class activation mapping) и хуки PyTorch, чтобы определить фокус решения модели о том, как подключить PyTorch к Google TensorBoard для отладки. Я покажу, как использовать флеймграфы для выявления узких мест в преобразованиях и обучающих конвейерах, а также приведу пример ускорения медленного преобразования. Наконец, мы рассмотрим, как обменять вычисления на память при работе с большими моделями, используя *контрольные точки*. Но сначала — пара слов о данных.

Три часа ночи. Что делают ваши данные?

Прежде чем окунуться в волшебный мир возможностей TensorBoard и градиентное создание контрольных точек, спросите себя: понимаете ли вы свои данные? Если вы классифицируете входные данные, есть ли у вас сбалансированная выборка по всем доступным маркировкам? Обучение, валидация и набор данных для теста?

И кроме того, вы уверены, что маркировки верны? Известно, что важные наборы данных на основе анализа изображений, такие как MNIST и CIFAR-10 (Канадский институт перспективных исследований), содер-

жат неправильные метки. Стоит проверить свои, особенно если категории схожи между собой, например породы собак или сорта растений. Простая проверка правильности ваших данных может в конечном итоге сэкономить много времени, например, если вы обнаружите, что одна категория маркировок включает только крошечные изображения, тогда как у всех остальных категорий есть примеры с большим разрешением.

Итак, если вы убедились, что все данные в порядке, тогда давайте приступим к TensorBoard и проверим модель на наличие некоторых возможных проблем.

TensorBoard

TensorBoard — это веб-приложение, предназначенное для визуализации различных аспектов нейронных сетей. Оно позволяет просматривать статистику в реальном времени, например, такие параметры, как точность, значения активации и потерь и все, что вы хотите отправить по проводной сети. Хотя оно было написано с оглядкой на TensorFlow, у него довольно простой API, поэтому работа с ним в PyTorch ничем не отличается. Давайте установим и посмотрим, как его использовать, чтобы получить представление о наших моделях.

Читая о PyTorch, вы, скорее всего, встретите ссылки на приложение под названием Visdom (https://oreil.ly/rZqv2), которое является альтернативой TensorBoard от Facebook. До PyTorch v1.1 Visdom с PyTorch использовался для поддержки визуализаций, а другие библиотеки, такие как `tensorboardX`, можно было интегрировать с Tensor Board. Несмотря на то что поддержка Visdom не приостановлена, включение официальной интеграции TensorBoard в версии 1.1 и выше предполагает, что разработчики PyTorch признали TensorBoard как де-факто инструмент визуализации нейронной сети.

Установка TensorBoard

Установить TensorBoard можно с помощью `pip` или `conda`:

```
pip install tensorboard
conda install tensorboard
```

Версия TensorBoard для PyTorch должна быть v1.14 или выше.

TensorBoard можно запустить через командную строку:

```
tensorboard --logdir=runs
```

Затем вы можете перейти по адресу http://[your-machine]:6006, где увидите экран приветствия (рис. 7.1). Теперь мы можем отправить данные в приложение.

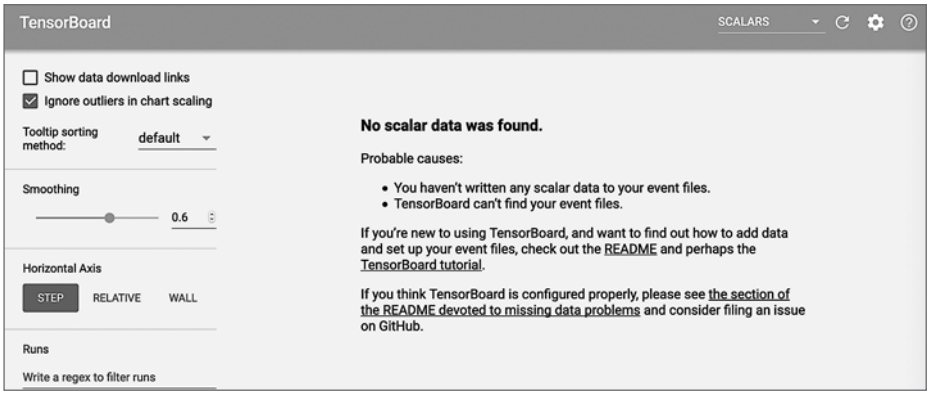

Рис. 7.1. TensorBoard

Отправка данных в TensorBoard

Модуль для использования TensorBoard с PyTorch находится в `torch.utils.tensor board`:

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter()
writer.add_scalar('example', 3)
```

Для связи с TensorBoard мы используем класс `SummaryWriter` и стандартную точку размещения регистрации выходных данных, *./runs*, но также можно отправить скаляр с помощью `add_scalar`. Поскольку `SummaryWriter` работает асинхронно, это может занять некоторое время, но вы должны увидеть обновление TensorBoard (рис. 7.2).

Не очень впечатляет, да? Давайте напишем цикл, который отправляет обновления из начальной точки:

```
import random
value = 10
writer.add_scalar('test_loop', value, 0)
for i in range(1,10000):
  value += random.random() - 0.5
  writer.add_scalar('test_loop', value, i)
```

Проходя цикл, TensorBoard строит график случайного блуждания, который мы делаем из 10 (рис. 7.3). Если мы снова запустим код, то увидим, что он сгенерировал другой *запуск* внутри дисплея, и в левой части вебстраницы можно выбрать, хотим мы посмотреть все прогоны или только некоторые.

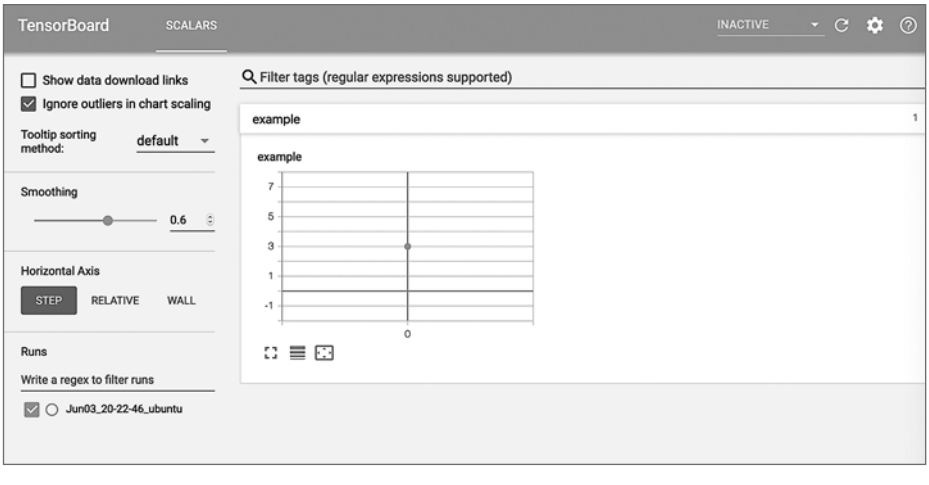

Рис. 7.2. Пример точки данных в TensorBoard

Мы можем использовать его, чтобы заменить операторы `print` в цикле обучения, и отправить саму модель, чтобы получить представление в TensorBoard.

```
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, transforms,models

writer = SummaryWriter()
model = models.resnet18(False)
writer.add_graph(model,torch.rand([1,3,224,224]))
def train(model, optimizer, loss_fn, train_data_loader, test_data_loader,
    epochs=20):
    model = model.train()
    iteration = 0

    for epoch in range(epochs):
        model.train()
        for batch in train_loader:
            optimizer.zero_grad()
            input, target = batch
            output = model(input)
            loss = loss_fn(output, target)
            writer.add_scalar('loss', loss, epoch)
            loss.backward()
            optimizer.step()

        model.eval()
        num_correct = 0
        num_examples = 0
        for batch in val_loader:
            input, target = batch
            output = model(input)
            correct = torch.eq(torch.max(F.softmax(output),
                      dim=1)[1], target).view(-1)
            num_correct += torch.sum(correct).item()
            num_examples += correct.shape[0]
            print("Epoch {}, accuracy = {:.2f}".format(epoch,
                   num_correct / num_examples)
            writer.add_scalar('accuracy', num_correct / num_examples, epoch)
        iterations += 1
```

Когда дело доходит до `add_graph()`, нужно отправить тензор для трассировки через модель, а также саму модель. Как только это произойдет, вы увидите, что в TensorBoard появится `GRAPHS` (рис. 7.4), а кликнув на большом блоке ResNet, вы увидите дополнительные детали структуры модели.

Рис. 7.3. График случайного блуждания в TensorBoard

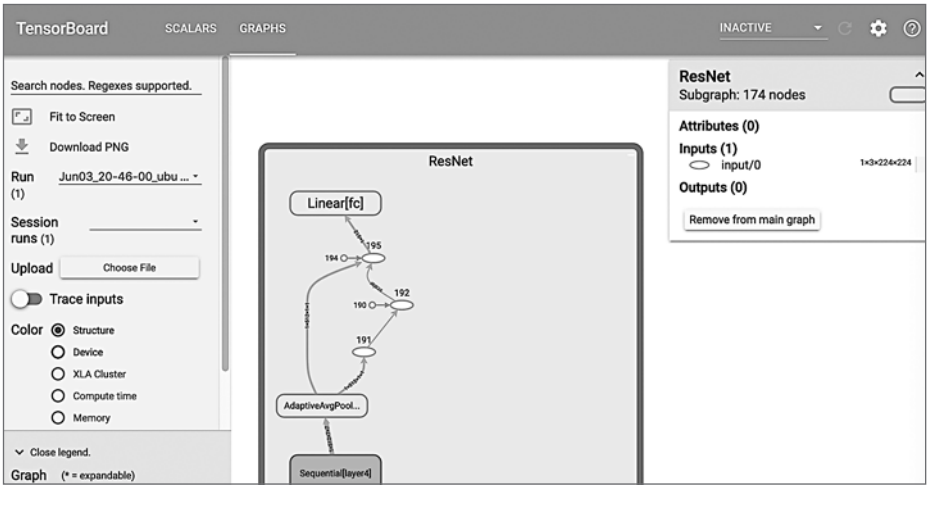

Рис. 7.4. Визуализация ResNet

Теперь у нас есть возможность отправлять информацию о точности и потерях, а также структуру модели в TensorBoard. Агрегируя несколько прогонов информации о точности и потерях, мы можем увидеть, отличается ли конкретный прогон от всех остальных, что может быть весьма полезным при выяснении причин, по которым обучающий прогон дал плохие результаты. Мы еще вернемся к TensorBoard, но сначала давайте рассмотрим другие признаки, которые можно отладить с помощью PyTorch.

Хуки PyTorch

В PyTorch есть хуки — функции, которые можно присоединить к тензору или модулю при прямом или обратном проходе. Когда во время прохода PyTorch встречает модуль с хуком, он вызывает зарегистрированные хуки. Зарегистрированный на тензоре хук будет вызываться при расчете его градиента.

Хуки — это потенциально сильные способы управления модулями и тензорами, так как при необходимости вы можете полностью заменить выходные данные того, что поступает в хук. Вы можете изменить градиент, замаскировать активацию, заменить все смещения в модуле и т. д. Однако в этой главе мы будем использовать их как способ получения информации о сети при прохождении данных.

В модели ResNet-18 мы можем прикрепить прямой хук к определенной части модели, используя `register_forward_hook`:

```
def print_hook(self, module, input, output):
  print(f"Shape of input is {input.shape}")

model = models.resnet18()
hook_ref= model.fc.register_forward_hook(print_hook)
model(torch.rand([1,3,224,224]))
hook_ref.remove()
model(torch.rand([1,3,224,224]))
```

Если вы запустите этот код, то увидите выведенный текст, показывающий форму ввода для слоя линейного классификатора модели. Обратите внимание, что во второй раз, когда вы пропускаете случайный тензор через модель, вы не должны видеть оператор `print`. Когда мы добавляем хук к модулю или тензору, PyTorch возвращает ссылку на этот хук. Мы всегда должны сохранять ее (здесь мы делаем это в `hook_ref`) и по окончании про-

цесса вызывать `remove()`. Если вы не сохраните ссылку, она будет просто висеть и занимать ценную память (и тратить вычислительные ресурсы во время передачи). Обратные хуки работают точно так же, за исключением того, что вы вместо этого вызываете `register_backward_hook()`.

Конечно, если мы можем выполнить команду `print()`, то можем и отправить ее в TensorBoard! Давайте посмотрим, как использовать хуки и TensorBoard для получения важной статистики на слоях во время обучения.

Построение графика среднего и стандартного отклонения

Для начала мы задаем функцию, которая будет отправлять среднее и стандартное отклонение выходного слоя в TensorBoard:

```
def send_stats(i, module, input, output):
  writer.add_scalar(f"{i}-mean",output.data.std())
  writer.add_scalar(f"{i}-stddev",output.data.std())
```

Мы не можем использовать этот код для настройки прямого хука, но с помощью функции `partial()` мы можем создать несколько прямых хуков, которые будут сами прикрепляться к слою с заданным значением `i`, что гарантирует передачу правильных значений на соответствующие графики в TensorBoard:

```
from functools import partial

for i,m in enumerate(model.children()):
  m.register_forward_hook(partial(send_stats, i))
```

Обратите внимание, что мы используем `model.children()`, который будет прикрепляться только к каждому верхнему блоку модели, поэтому если у нас есть слой `nn.Sequential()` (который будет также в модели на основе ResNet), то мы прикрепляем хук только к этому блоку, а не к блоку каждого отдельного модуля в списке `nn.Sequential`.

Если мы обучаем модель с обычной функцией обучения, мы должны увидеть, как активация начинает поступать в TensorBoard (рис. 7.5). В пользовательском интерфейсе вам придется переключиться на физическое время, поскольку мы больше не отправляем информацию о шагах обратно в TensorBoard с помощью хуков и получаем информацию о модуле только при вызове хука PyTorch.

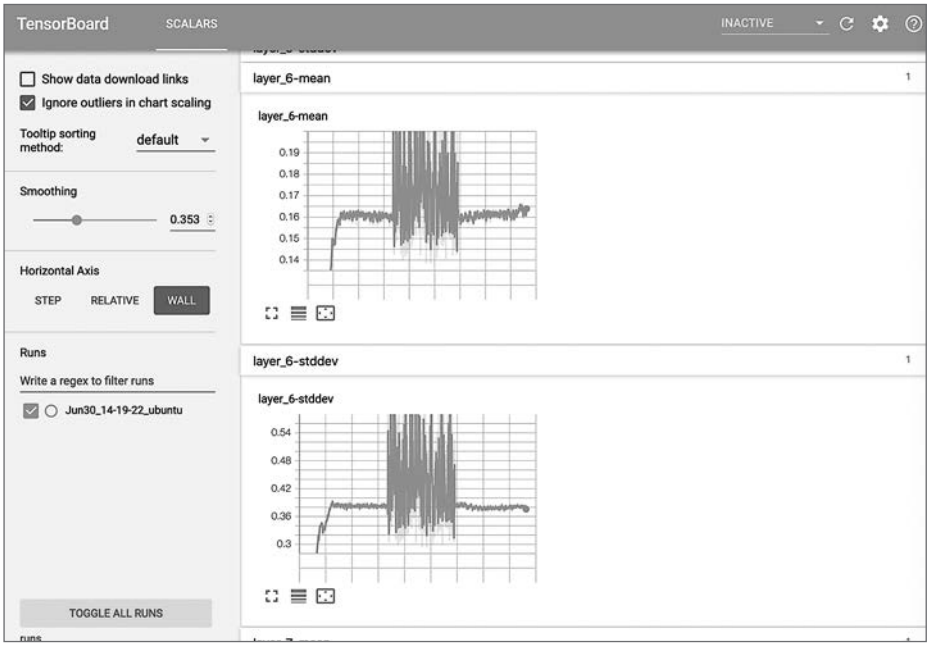

Рис. 7.5. Среднее и стандартное отклонение модулей в TensorBoard

В главе 2 я писал о том, что в идеале слои нейронной сети должны иметь среднее значение 0 и стандартное отклонение 1, чтобы убедиться, что наши вычисления не уходят в бесконечность или в ноль.

Посмотрите на слои в TensorBoard. Похоже ли, что они остаются в этом диапазоне значений? Есть ли на графике пики и спады? Если да, это может свидетельствовать о том, что сеть испытывает трудности при обучении. На рис. 7.5 среднее значение близко к нулю, но и стандартное отклонение также довольно близко к нулю. Если это происходит на многих уровнях вашей сети, это может быть признаком того, что ваши функции активации (например, `ReLU`) не совсем подходят для проблемного домена. Возможно, стоит поэкспериментировать с другими функциями, чтобы понять, улучшают ли они производительность модели; `LeakyReLU` от PyTorch — это хорошая альтернатива, предлагающая аналогичные активации для стандартной `ReLU`, но позволяющая получить больше информации, что может способствовать обучению.

На этом мы завершаем обзор TensorBoard. Если у вас остались вопросы, ознакомьтесь с дополнительными источниками в конце главы. А пока давайте разберемся, как заставить модель объяснить, каким образом она пришла к конкретному решению.

Карты активаций класса

Карты активаций класса (*Class activation mapping,* CAM) — это метод визуализации активаций сети после классификации входящего тензора. В классификаторах на основе анализа изображений они часто отображаются в виде тепловой карты поверх исходного изображения, как показано на рис. 7.6.

На основании тепловой карты мы можем сделать вывод о том, как сеть выбрала персидскую кошку из доступных классов ImageNet. Наиболее высокие активации сети наблюдаются вокруг морды и тела кошки, в остальных частях изображения активации низкие.

Чтобы сгенерировать тепловую карту, мы захватываем активации последнего слоя свертки сети непосредственно перед тем, как он перейдет в слой `Linear`, поскольку так мы можем увидеть, что объединенные слои CNN считают важными, когда мы переходим к финальному преобразованию из изображения в класс.

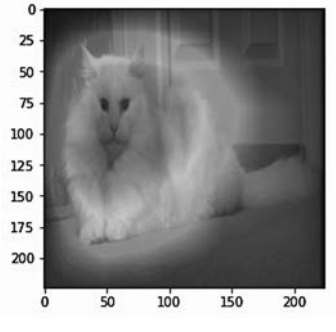

Рис. 7.6. Карты активаций класса с выделением

Благодаря функции хуков PyTorch это довольно просто. Заворачиваем хук в класс `SaveActivations`:

```
class SaveActivations():
    activations=None
    def __init__(self, m):
      self.hook = m.register_forward_hook(self.hook_fn)
    def hook_fn(self, module, input, output):
      self.features = output.data
    def remove(self):
      self.hook.remove()
```

Затем мы проталкиваем наше изображение через сеть (нормализуя для ImageNet), применяем `softmax`, чтобы преобразовать выходной тензор в вероятности, и используем `torch.topk()` для извлечения как максимальной вероятности, так и ее индекса:

```
import torch
from torchvision import models, transforms
from torch.nn import functional as F

casper = Image.open("casper.jpg")
# Imagenet mean/std

normalize = transforms.Normalize(
   mean=[0.485, 0.456, 0.406],
   std=[0.229, 0.224, 0.225]
)

preprocess = transforms.Compose([
   transforms.Resize((224,224)),
   transforms.ToTensor(),
   normalize
])

display_transform = transforms.Compose([
   transforms.Resize((224,224))])

casper_tensor = preprocess(casper)

model = models.resnet18(pretrained=True)
model.eval()
casper_activations = SaveActivations(model.layer_4)
prediction = model(casper_tensor.unsqueeze(0))
pred_probabilities = F.softmax(prediction).data.squeeze()
casper_activations.remove()
torch.topk(pred_probabilities,1)
```

Я еще не объяснил, что такое `torch.nn.functional`. Лучше всего воспринимать его как нечто содержащее реализацию *функций*, представленных в `torch.nn`. Например, если вы создаете экземпляр `torch.nn.softmax()`, то вы получаете объект с методом `forward()`, который выполняет `softmax`. Если вы посмотрите на фактический источник `torch.nn.softmax()`, то увидите, что все, что делает этот метод, — это вызывает `F.softmax()`. Поскольку в данном случае нам не нужно, чтобы `softmax` был частью сети, мы просто вызываем базовую функцию.

Если мы зайдем в `casper_activations.activations`, то увидим, что он заполнен тензором, который содержит активации последнего требуемого слоя свертки. Затем мы делаем следующее:

```
fts = sf[0].features[idx]
        prob = np.exp(to_np(log_prob))
        preds = np.argmax(prob[idx])
        fts_np = to_np(fts)
        f2=np.dot(np.rollaxis(fts_np,0,3), prob[idx])
        f2-=f2.min()
        f2/=f2.max()
        f2
plt.imshow(dx)
plt.imshow(scipy.misc.imresize(f2, dx.shape), alpha=0.5, cmap='jet');
```

Так мы вычисляем скалярное произведение активаций Карты (помните, что мы индексируем в 0 из-за пакетирования входного тензора в первой размерности). Как я писал в главе 1, PyTorch хранит данные изображения в формате C × H × W, поэтому для показа изображения нам нужно изменить размеры обратно на H × W × C.

Затем мы удаляем минимумы из тензора и масштабируем по максимуму, чтобы гарантировать, что мы фокусируемся только на самых высоких активациях в полученной тепловой карте (то есть на том, что говорит в пользу персидской кошки).

Наконец, мы используем чудесные возможности `matplot`, чтобы отобразить выделение теплокарты, а затем тензор сверху, с измененным размером и стандартной цветовой картой `jet`. Обратите внимание, что, заменив `idx` другим классом, вы можете увидеть тепловую карту, показывающую, какие активации (если таковые имеются) присутствуют на изображении

при классификации. Если модель спрогнозирует автомобиль, вы можете увидеть, какие части изображения были использованы для принятия этого решения. На втором месте по вероятности находится ангорский кролик, и мы видим на основании CAM, что сеть сфокусирована на его пушистой шерстке!

Теперь мы знаем, что делает модель, когда принимает решение. На этом остановимся и выясним, на что модель тратит большую часть времени, пока находится в цикле обучения или когда принимает решения.

Флеймграфы

В отличие от TensorBoard, *флеймграфы* не были созданы специально для нейронных сетей. И даже не для TensorFlow. Фактически флеймграфы появились еще до 2011 года, когда инженер Брендан Грегг из компании Joyent придумал технику отладки для задачи в MySQL. Идея заключалась в том, чтобы взять массивные стектрейсы и преобразовать их в одно изображение, которое само по себе покажет то, что происходит в CPU в течение определенного периода времени.

Сейчас Брендан Грегг работает в Netflix. Он опубликовал огромное количество работ, связанных с производительностью системы. Почитать их можно на его сайте http://www.brendangregg.com/.

Используя пример вставки строки в таблицу в MySQL, мы семплируем стеки сотни или тысячи раз в секунду. Каждый раз мы получаем *стектрейс*, который показывает нам все функции в стеке на тот момент времени. Поэтому если мы находимся в функции, которая была вызвана другой функцией, то получим трассировку, которая включает в себя функции как вызываемого, так и вызывающего. Пример трассировки выглядит так:

```
65.00%  0.00% mysqld  [kernel.kallsyms]  [k] entry_SYSCALL_64_fastpath
          |
          ---entry_SYSCALL_64_fastpath
             |
             |--18.75%-- sys_io_getevents
             |          read_events
```

```
|          schedule
|          __schedule
|          finish_task_switch
|
|--10.00%-- sys_fsync
|          do_fsync
|          vfs_fsync_range
|          ext4_sync_file
|          |
|          |--8.75%-- jbd2_complete_transaction
|          |          jbd2_log_wait_commit
|          |          |
|          |          |--6.25%-- _cond_resched
|          |          |          preempt_schedule_common
|          |          |          __schedule
```

Этой информации там *много*; здесь всего лишь крошечный пример трассировки стека размером 400 КБ. Даже при таком сравнении (которое может присутствовать не во всех стектрейсах) трудно разобраться, что здесь творится.

Из рис. 7.7 видно, что версия с использованием флеймграфа все же проста и понятна. Ось y — это высота стека, а ось x, хотя и не обозначает время, является представлением того, как часто эта функция находится в стеке после семплирования. Если бы у нас была функция в верхней части стека, которая покрывала бы, скажем, 80 % графика, мы бы знали, что программа тратит чертовски много времени на эту функцию и что, возможно, следует проверить функцию, чтобы понять, почему все происходит так долго.

Вы можете спросить: «Какое это имеет отношение к глубокому обучению?» Весьма справедливо; в исследованиях по глубокому обучению это обычное явление: когда обучение замедляется, вы просто покупаете еще 10 графических процессоров или платите Google за тензорные процессоры. Но порой дело не в GPU. Возможно, действительно преобразование проходит слишком медленно и новые блестящие графические адаптеры не работают так, как хотелось бы.

На первый взгляд флеймграфы предоставляют простой способ выявления узких мест, связанных с процессором, и часто встречаются в практических решениях задач глубокого обучения. Например, вспомните все те графические преобразования, о которых мы говорили в главе 4. Большинство из них используют библиотеку Python Imaging Library и полностью зависят от возможностей процессора. Работая с большими наборами данных, вы будете выполнять эти преобразования снова и снова в рамках цикла обучения!

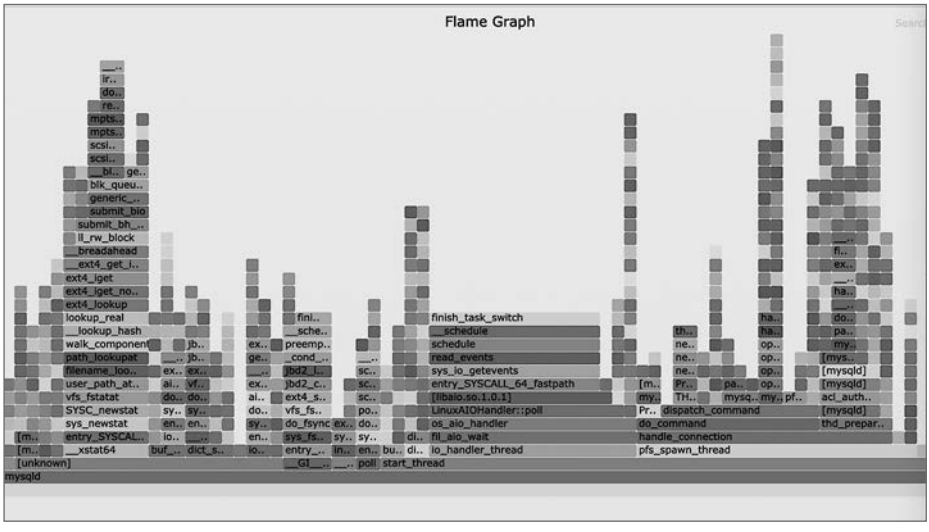

Рис. 7.7. Флеймграф MySQL

Хотя о них не часто вспоминают в контексте глубокого обучения, флеймграфы — отличный инструмент, который стоит иметь в виду. По крайней мере, можно показать их своему боссу и сказать, что ваша работа действительно зависит от графического процессора и к следующему четвергу — кровь из носу — вам нужны тензорные процессоры! А пока рассмотрим получение флеймграфов из циклов обучения, а также решение проблемы медленного преобразования путем перемещения из CPU в GPU.

Установка py-spy

Есть много способов сгенерировать стектрейсы, которые можно преобразовать в флеймграфы. Тот, о котором я говорил в предыдущем разделе, был создан с помощью сложного и эффективного инструмента Linux Perf. Мы пойдем более простым путем и используем `py-spy`, стек-профайлер на базе Rust для генерации флеймграфов напрямую. Установите его через `pip`:

```
pip install py-spy
```

Найдите идентификатор (PID) запущенного процесса и присоедините `py-spy` с помощью аргумента `--pid`:

```
py-spy --flame profile.svg --pid 12345
```

Или передайте скрипт Python. Для начала давайте запустим его на простом скрипте Python:

```
import torch
import torchvision

def get_model():
    return torchvision.models.resnet18(pretrained=True)

def get_pred(model):
    return model(torch.rand([1,3,224,224]))

model = get_model()

for i in range(1,10000):
    get_pred(model)
```

Сохраните код как *flametest.py* и запустите на нем `py-spy`, семплируя 99 раз в секунду и прогоняя цикл 30 секунд:

```
py-spy -r 99 -d 30 --flame profile.svg -- python t.py
```

Откройте файл *profile.svg* в браузере и посмотрите на полученный график.

Чтение флеймграфов

На рис. 7.8 показано, как примерно должен выглядеть график (из-за семплирования он может выглядеть несколько иначе). Сразу бросается в глаза, что график идет вниз, а не вверх. `py-spy` записывает флеймграфы в формате *icicle*, поэтому стек похож на сталактиты, а не на классический флеймграф. Мне больше нравится стандартный формат, но `py-spy` не дает изменять его. В целом это не так уж важно.

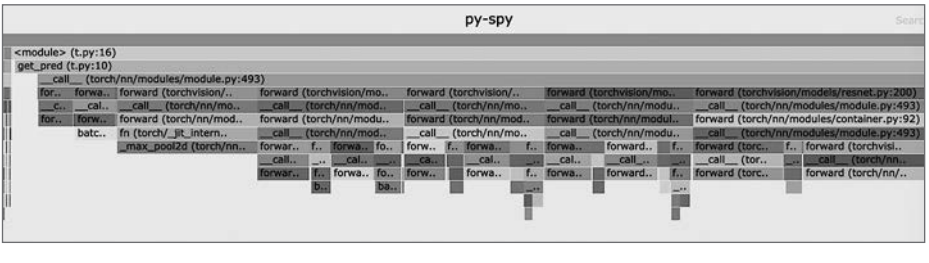

Рис. 7.8. Флеймграф на ResNet. Загрузка и вывод

Видно, что большая часть времени тратится на различные вызовы `forward()`, и это логично, так как мы делаем много прогнозов. А что насчет тех крошечных блоков слева? Если вы кликнете на них, то увидите, что файл SVG увеличится (рис. 7.9).

Здесь мы видим скрипт, устанавливающий модуль ResNet-18, а также вызывающий `load_state_dict()` для загрузки сохраненных весов с диска (он вызывался через `pretrained=True`). Кликните на Reset Zoom, чтобы вернуться к полномасштабному флеймграфу. Панель поиска справа выделит соответствующие строки фиолетовым цветом, если вы попытаетесь найти функцию.

Попробуйте сделать это с помощью *resnet*, и он покажет каждый вызов функции в стеке, где в имени есть *resnet*. Это может быть полезно для нахождения небольшого количества функций в стеке или для просмотра того, как часто этот паттерн отображается на графике.

Поиграйте немного с SVG и посмотрите, сколько времени у CPU отнимают такие штуки, как, например, Batch-Norm и субдискретизация. Далее мы рассмотрим способ использования флеймграфов, благодаря которому можно найти проблему, исправить ее и выполнить верификацию с помощью другого флеймграфа.

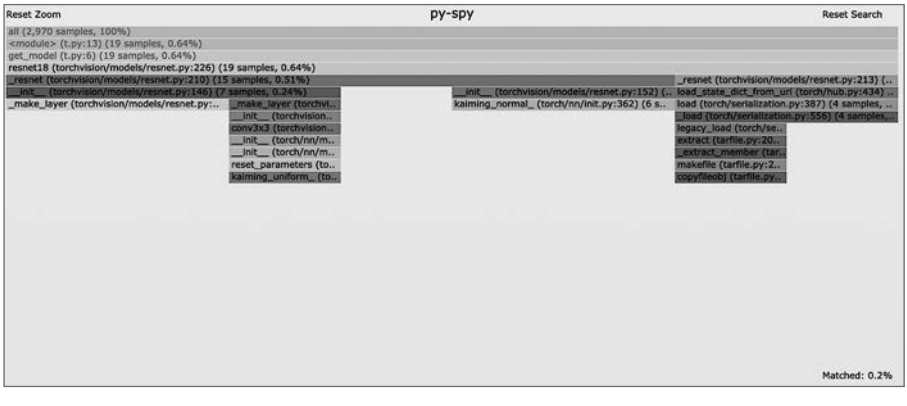

Рис. 7.9. Детализированный флеймграф

Решение задачи медленного преобразования

В реальных ситуациях часть конвейера данных может привести к медленной работе. Эта проблема заслуживает особого внимания, так как медленное преобразование во время обучения происходит часто, вызывая огромные сложности при создании модели. Вот пример конвейера преобразования и загрузчика данных:

```
import torch
import torchvision
from torch import optim
import torch.nn as nn
from torchvision import datasets, transforms, models
import torch.utils.data
from PIL import Image
import numpy as np

device = "cuda:0"
model = models.resnet18(pretrained=True)
model.to(device)


class BadRandom(object):
    def __call__(self, img):
        img_np = np.array(img)
        random = np.random.random_sample(img_np.shape)
        out_np = img_np + random
        out = Image.fromarray(out_np.astype('uint8'), 'RGB')
        return out

    def __repr__(self):
        str = f"{self.__class__.__name__}"
        return str

train_data_path = "catfish/train"
image_transforms =
torchvision.transforms.Compose(
  [transforms.Resize((224,224)),BadRandom(), transforms.ToTensor()])
```

Мы не собираемся проводить полный цикл обучения; вместо этого мы имитируем 10 эпох, просто извлекая изображения из загрузчика обучающих данных:

```
train_data = torchvision.datasets.ImageFolder(root=train_data_path,
transform=image_transforms)
batch_size=32
train_data_loader = torch.utils.data.DataLoader(train_data,
batch_size=batch_size)
```

```
optimizer = optim.Adam(model.parameters(), lr=2e-2)
criterion = nn.CrossEntropyLoss()

def train(model, optimizer, loss_fn, train_loader, val_loader,
epochs=20, device='cuda:0'):
    model.to(device)
    for epoch in range(epochs):
        print(f"epoch {epoch}")
        model.train()
        for batch in train_loader:
            optimizer.zero_grad()
            ww, target = batch
            ww = ww.to(device)
            target= target.to(device)
            output = model(ww)
            loss = loss_fn(output, target)
            loss.backward()
            optimizer.step()

        model.eval()
        num_correct = 0
        num_examples = 0
        for batch in val_loader:
            input, target = batch
            input = input.to(device)
            target= target.to(device)
            output = model(input)
            correct = torch.eq(torch.max(output, dim=1)[1], target).view(-1)
            num_correct += torch.sum(correct).item()
            num_examples += correct.shape[0]
        print("Epoch {}, accuracy = {:.2f}"
        .format(epoch, num_correct / num_examples))

train(model,optimizer,criterion,
train_data_loader,train_data_loader,epochs=10)
```

Давайте запустим этот код через `py-spy`, как мы уже делали:

```
py-spy -r 99 -d 120 --flame slowloader.svg -- python slowloader.py
```

Если вы откроете получившийся файл *slowloader.svg*, то увидите нечто похожее на рис. 7.10. Хотя флеймграф в основном занят загрузкой изображений и преобразованием их в тензоры, тратится 16,87 % времени семплирования при применении случайного шума. Реализация `BadRandom` применяет шум на этапе PIL, а не на этапе тензора, поэтому мы используем библиотеку изображений и NumPy, а не сам PyTorch. Таким образом, первой идеей, вероятно, будет переписать преобразование

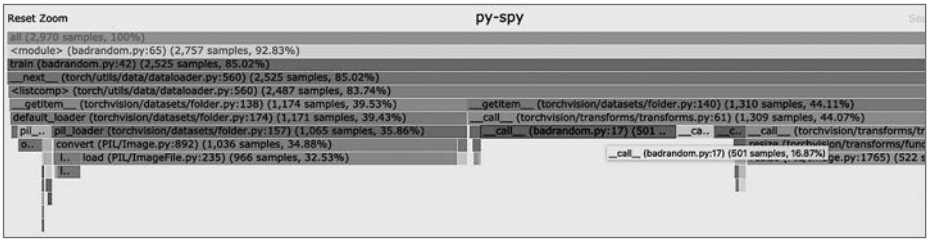

Рис. 7.10. Флеймграф с BadRandom

так, чтобы оно работало с тензорами, а не с изображениями PIL. Обратите внимание, что при изменении производительности важно всегда все как следует измерить.

Есть одна любопытная штука, которая присутствовала с самого начала книги, хотя я до сих пор и не обращал на нее внимания: вы заметили, что мы извлекаем пакеты из загрузчика данных и затем помещаем их в GPU? Поскольку преобразования происходят, когда загрузчик получает пакеты из класса набора данных, эти преобразования всегда будут происходить в CPU. В некоторых случаях это может привести к сумасшедшему латеральному мышлению. Мы применяем случайный шум к каждому изображению. А что, если бы мы могли применить случайный шум к каждому изображению одновременно?

На первый взгляд это может показаться сложным: мы добавляем случайный шум к изображению. Мы можем записать это как $x + y$, где x — наше изображение, а y — шум. Мы знаем, что изображение и шум являются трехмерными (ширина, высота, каналы), поэтому все, что мы делаем, — это умножаем матрицы. И в пакете мы будем делать это z раз. Мы просто перебираем каждое изображение, когда извлекаем его из загрузчика. Но учтите, что в конце процесса загрузки изображения преобразуются в тензоры, пакеты $[z, c, h, w]$. Можно ли просто добавить случайный тензор формы $[z, c, h, w]$ и применить случайный шум таким образом? Шум применяется не последовательно, а одновременно. Теперь у нас есть матричная операция и очень дорогой графический процессор, который неплохо справляется с матричными операциями. Попробуйте проделать то же самое в Jupyter Notebook, чтобы увидеть разницу между тензорными матричными операциями графического и центрального процессоров:

```
cpu_t1 = torch.rand(64,3,224,224)
cpu_t2 = torch.rand(64,3,224,224)
%timeit cpu_t1 + cpu_t2
>> 5.39 ms ± 4.29 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

gpu_t1 = torch.rand(64,3,224,224).to("cuda")
gpu_t2 = torch.rand(64,3,224,224).to("cuda")
%timeit gpu_t1 + gpu_t2
>> 297 µs ± 338 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Это почти в 20 раз быстрее. Мы можем не выполнять это преобразование в загрузчике данных, а выполнить матричные операции после того, как получим весь пакет:

```
def add_noise_gpu(tensor, device):
  random_noise = torch_rand_like(tensor).to(device)
  return tensor.add_(random_noise)
```

Добавьте эту строку в цикле обучения после `input.to(device)`:

```
input = add_noise_gpu(input, device)
```

Затем удалите преобразование `BadRandom` из конвейера преобразования и протестируйте снова с помощью py-spy. Новый флеймграф показан на рис. 7.11. Все происходит так быстро, что процесс больше не отображается даже при нашей частоте семплинга. Мы только что ускорили разработку кода почти на 17 %! Теперь для графического процессора можно написать не все стандартные преобразования, но если это возможно и преобразование замедляет работу, то это определенно стоит учесть.

Теперь, когда мы рассмотрели вычисления, пришло время взглянуть на другую очевидную вещь — память. В частности, память графических процессоров.

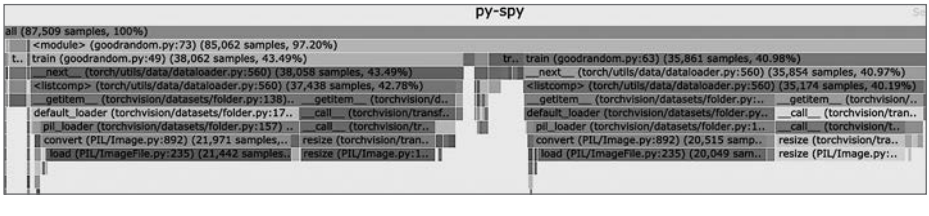

Рис. 7.11. Флеймграф со случайным шумом с ускорением вычислений на графических процессорах

Отладка проблем с графическим процессором

В этом разделе мы подробно рассмотрим сам графический процессор. Скоро вы узнаете, что при обучении больших моделей ваш блестящий графический процессор, на который вы потратили уйму денег (или, что более разумно, привязали его к облачному экземпляру), регулярно падает на колени, горько жалуясь на нехватку памяти. Но у него есть целые гигабайтища памяти! Как же они могут закончиться?

Модели, как правило, высасывают много памяти. Например, у ResNet-152 примерно 60 миллионов активаций, и все они занимают драгоценное место в памяти графического процессора. Давайте разберемся, как заглянуть внутрь него, чтобы определить, что может происходить, если у вас мало памяти.

Проверка графического процессора

Предполагая, что вы используете графический процессор NVIDIA (проверьте веб-сайт драйверов вашего альтернативного поставщика графического процессора на наличие собственных утилит, если вы используете что-то другое), установка CUDA включает довольно полезный инструмент командной строки — `nvidia-smi`.

При запуске без аргументов этот инструмент может дать вам снаппшот памяти, используемой на графическом процессоре, и даже еще лучше — он может показать, как именно она используется! На рис. 7.12 показан результат запуска `nvidia-smi` в терминале. В блокноте вы можете вызвать утилиту с помощью `!nvidia-smi`.

Это пример моего личного домашнего оборудования, работающего на 1080 Ti. Я использую несколько блокнотов, каждый из которых занимает часть памяти, но один занимает целых 4 ГБ! Вы можете получить текущий PID блокнота, используя `os.getpid()`. Оказывается, процесс, использующий больше всего памяти, на самом деле был экспериментальным блокнотом, который я использовал для тестирования преобразований графического процессора в предыдущем разделе! Представьте себе, как быстро заканчивается память, которую занимает модель, пакетные данные и данные для прямого и обратного проходов.

```
ian@ubuntu:~/notebooks$ nvidia-smi
Fri Jun  7 10:27:32 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.54                 Driver Version: 396.54                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 108...  Off  | 00000000:01:00.0  On |                  N/A |
|  0%   26C    P8     9W / 250W |   8079MiB / 11176MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      2006      G   /usr/lib/xorg/Xorg                            32MiB |
|    0      2413      G   /usr/bin/gnome-shell                          58MiB |
|    0      3993      C   /home/ian/anaconda3/bin/python              1407MiB |
|    0     17301      C   /home/ian/anaconda3/bin/python               527MiB |
|    0     19205      C   /home/ian/anaconda3/bin/python               523MiB |
|    0     31226      C   /home/ian/anaconda3/bin/python               885MiB |
|    0     32113      C   /home/ian/anaconda3/bin/python              4633MiB |
+-----------------------------------------------------------------------------+
```

Рис. 7.12. Результаты nvidia-smi

У меня также есть несколько запущенных процессов, которые, что удивительно, отвечают за графику, а именно X-сервер и GNOME. Если вы не создали локальную машину, то, скорее всего, не увидите их.

Кроме того, PyTorch выделяет часть памяти для себя и CUDA на процесс, который занимает около 0,5 ГБ памяти.

Это означает, что лучше работать с одним проектом за раз и не оставлять Jupyter Notebook работать над всем сразу, как это делал я (можно использовать меню Kernel, чтобы завершить процесс Python, подключенный к блокноту).

Запуск `nvidia-smi` сам по себе даст текущий снапшот использования графического процессора, но вы можете получить непрерывные выходные данные, используя флаг `-l`. Вот пример команды, которая будет сбрасы-

вать отметку времени, использованную память, свободную память, общий объем памяти и использование графического процессора каждые 5 секунд:

```
nvidia-smi --query-gpu=timestamp,
memory.used, memory.free,memory.total,utilization.gpu --format=csv -l 5
```

Если вы действительно думаете, что ваш графический процессор использует больше памяти, чем следовало бы, можете попробовать подключить программу чистки памяти в Python. Если у вас есть `tensor_to_be_deleted`, он вам больше не нужен и вы хотите удалить его из графического процессора, то воспользуйтесь библиотекой fast.ai library и командой `del`:

```
import gc
del tensor_to_be_deleted
gc.collect()
```

Если вы много работаете над созданием и воссозданием моделей в Jupyter Notebook, вы могли заметить, что удаление некоторых ссылок и запуск программы чистки памяти с помощью `gc.collect()` приводит к восстановлению некоторой части памяти. Если проблемы с памятью остаются, продолжайте читать, потому что дальше вы найдете решение!

Градиентное создание контрольных точек

Несмотря на все хитрости удаления лишнего и программы для очистки памяти, о которых я рассказал в предыдущем разделе, памяти все равно может не хватить. Следующий шаг для большинства приложений — уменьшение размера пакета данных, проходящих через модель во время цикла обучения. Это сработает, но вы будете увеличивать время обучения для каждой эпохи, и вполне вероятно, что модель не будет такой же хорошей, как аналогичная, обученная с достаточным объемом памяти, которого хватает для обработки больших пакетов данных, поскольку вы будете видеть больше наборов данных на каждом прогоне. Однако для больших моделей в PyTorch мы можем обменять компьютерные вычисления на память, используя *градиентное создание контрольных точек*.

Одна из проблем при работе с большими моделями заключается в том, что прямой и обратный проход создают много промежуточных состояний, каждое из которых занимает память графического процессора. Цель градиентного создания контрольных точек состоит в уменьшении

количества состояний, которые могут быть в GPU путем *сегментирования* модели.

То есть при несегментированной модели размер пакета может быть больше в 4–10 раз, причем это компенсируется обучением, требующим значительных вычислительных ресурсов. Во время прямого прохода PyTorch сохраняет входные данные и параметры в сегмент, но при этом не выполняет сам прямой проход. Во время обратного прохода PyTorch их извлекает, и прямой проход вычисляется для этого сегмента.

Промежуточные значения передаются на следующий сегмент, но должны выполняться только на базе segment-by-segment.

Разделение модели на сегменты выполняется через `torch.utils.checkpoint.checkpoint_sequential()`. Он работает со слоями `nn.Sequential` или сгенерированными списками слоев, при условии, что они должны быть в той же последовательности, что и в модели. Вот как это будет работать с модулем `features` в AlexNet:

```
from torch.utils.checkpoint import checkpoint_sequential
import torch.nn as nn

class CheckpointedAlexNet(nn.Module):

    def __init__(self, num_classes=1000, chunks=2):
        super(CheckpointedAlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
```

```
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )
    def forward(self, x):
        x = checkpoint_sequential(self.features, chunks, x)
        x = self.avgpool(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x
```

Как видите, мало что изменилось. Создание контрольных точек становится простым дополнением к моделям, когда это необходимо. Мы добавили параметр `chunks` в новую версию модели: по умолчанию он разделен на два сегмента.

Все, что осталось сделать, — это вызвать `checkpoint_sequential` с модулем `features`, количеством сегментов и входными данными. Вот и все!

Обратите внимание: создание контрольных точек плохо работает со слоями `BatchNorm` или `Drop out`, так как они взаимодействуют с прямым проходом.

Чтобы обойти это, вы можете просто создать контрольные точки части модели до и после этих слоев. В `Check pointedAlexNet` мы, возможно, могли бы разбить модуль `classifier` на две части: одну, содержащую слои `Dropout`, без контрольных точек, и конечный модуль `nn.Sequential`, содержащий слои `Linear`; устанавливать контрольные точки которых мы могли бы так же, как в случае с `features`.

Если нужно уменьшить размер партий для запуска модели, подумайте о контрольных точках, а не о большем графическом процессоре!

Заключение

Надеюсь, что теперь вы как следует вооружены, если вдруг в обучении модели что-то пошло не так. В вашем распоряжении теперь есть множество инструментов, от очистки данных до запуска флеймграфов или визуализации TensorBoard. Вы также узнали о том, как с помощью контрольных точек можно поменять память на вычисления с преобразованиями на GPU, и наоборот.

Имея должным образом обученную и отлаженную модель, мы находимся на пути к самому сложному этапу: *эксплуатации модели в рабочей среде,* или попросту *продакшн*.

Дополнительные источники

- Документация TensorBoard, https://oreil.ly/MELKl
- TensorBoard GitHub, https://oreil.ly/21bIM
- Fast.ai, Урок 10: Looking Inside The Model, https://oreil.ly/K4dz-
- Investigation into BatchNorm within a ResNet model, https://oreil.ly/EXdK3
- Глубокое погружение в создание флеймграфов с Бренданом Греггом, https://oreil.ly/4Ectg
- Документация nvidia-smi, https://oreil.ly/W1g0n
- Документация по градиентному созданию контрольных точек PyTorch, https://oreil.ly/v0apy

ГЛАВА 8

PyTorch в рабочей среде

Теперь, когда вы знаете, как применять PyTorch для классификации изображений, текста и звука, следующий шаг — разобраться, как разворачивать PyTorch в рабочей среде. В этой главе мы создадим приложения, которые запускают вывод на моделях PyTorch через HTTP и gRPC. Затем упакуем эти приложения в контейнеры Docker и развернем их в кластер Kubernetes на Google Cloud.

Во второй части главы мы рассмотрим TorchScript — новую технологию, представленную в PyTorch 1.0 и позволяющую использовать трассировку «на лету» (JIT) для создания оптимизированных моделей, которые можно запускать на C ++. Мы также рассмотрим, как сжимать модели посредством квантования. Прежде всего, давайте обсудим обслуживание модели.

Обслуживание модели

Предыдущие шесть глав были посвящены созданию моделей в PyTorch, но создание модели — это лишь часть создания приложения для глубокого обучения. В конце концов, модель может иметь поразительную точность (или другую соответствующую метрику), но если она не делает никаких прогнозов, то зачем она вообще нужна? Требуется простой способ упаковывать модели, чтобы они могли отвечать на запросы (онлайн или как-то по-другому) и чтобы их можно было запустить с минимальными усилиями.

К счастью, Python позволяет быстро запустить веб-сервис с помощью фреймворка Flask. В этом разделе мы создадим простой сервис, который загружает нашу модель *кошка или рыбка* на базе ResNet, принимает запросы с URL-адресом изображения и возвращает ответ JSON, указывающий, кто на изображении — кошка или рыбка.

Что произойдет, если мы отправим модели фотографию собаки? Модель скажет, что это либо рыбка, либо кошка. Она больше ничего не знает, кроме доступных вариантов, и всегда выбирает один из них. Некоторые специалисты по глубокому обучению добавляют во время обучения дополнительный класс, `Unknown`, и маркированные примеры, которые не принадлежат ни одному из классов. В определенной степени это работает, но, по сути, так мы пытаемся заставить нейронную сеть выучить *все, что не является кошкой или рыбкой,* что трудно даже нам с вами, не говоря уже о серии матричных вычислений! Другой вариант — посмотреть на вероятность выходных данных, генерируемых окончательным `softmax`. Если прогноз модели составляет примерно 50/50 или разбросан по классам, то есть смысл подумать об использовании варианта `Unknown`.

Построение сервиса Flask

Давайте получим версию нашей модели с поддержкой веб-сервисов. *Flask* — это известная платформа для создания веб-сервисов с помощью Python, и в этой главе мы возьмем ее за основу. Установите библиотеку Flask с помощью `pip` или `conda`:

```
conda install -c anaconda flask
pip install flask
```

Создайте новый каталог с именем *catfish* и скопируйте определение модели как *model.py*:

```
from torchvision import models

CatfishClasses = ["cat","fish"]

CatfishModel = models.ResNet50()
CatfishModel.fc = nn.Sequential(nn.Linear(transfer_model.fc.in_features,500),
                  nn.ReLU(),
                  nn.Dropout(), nn.Linear(500,2))
```

Обратите внимание, что здесь мы не указываем предварительно обученную модель, так как будем загружать сохраненные веса в процессе запуска

сервера Flask. Создайте еще один скрипт Python, *catfish_server.py*. В нем мы запустим веб-сервис:

```
from flask import Flask, jsonify
from . import CatfishModel
from torchvision import transforms
import torch
import os

def load_model():
  return model

app = Flask(__name__)

@app.route("/")
def status():
  return jsonify({"status": "ok"})

@app.route("/predict", methods=['GET', 'POST'])
def predict():
  img_url = request.image_url
  img_tensor = open_image(BytesIO(response.content))
  prediction = model(img_tensor)
  predicted_class = CatfishClasses[torch.argmax(prediction)]
  return jsonify({"image": img_url, "prediction": predicted_class})

if __name__ == '__main__':
  app.run(host=os.environ["CATFISH_HOST"], port=os.environ["CATFISH_PORT"])
```

Можно запустить веб-сервер через командную строку, установив переменные среды `CATFISH_HOST` и CATFISH_PORT:

```
CATFISH_HOST=127.0.0.1 CATFISH_PORT=8080 python catfish_server.py
```

Если вы указываете веб-браузеру на http://127.0.0.1:8080, то должны получить ответ JSON `status:"ok"`, как показано на рис. 8.1.

Более подробно я раскрою тему дальше. Не развертывайте сервис Flask непосредственно в рабочую среду — встроенный сервер для нее не годится.

Чтобы сделать прогноз, найдите URL-адрес изображения и отправьте его в виде запроса `GET` с параметром `image_url` в путь `/predict`. Вы увидите ответ JSON, показывающий URL и прогнозируемый класс (рис. 8.2).

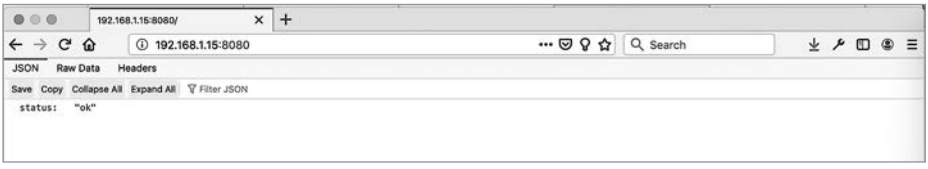

Рис. 8.1. Ответ OK от CATFISH

Сила Flask заключается в аннотациях `@app.route()`. Благодаря им мы можем присоединять обычные функции Python, которые будут запускаться, когда пользователь попадает в конкретную конечную точку. В методе `predict()` мы достаем параметр `img_url` из HTTP-запроса `GET` или `POST`, открываем этот URL-адрес как PIL-изображение и проталкиваем его через простой конвейер преобразования `torchvision`, чтобы изменить размер и преобразовать изображение в тензор. Получаем тензор в форме `[3,224,224]`, но ввиду особенностей нашей модели мы должны преобразовать ее в пакет размером 1, то есть `[1,3,224,224]`. Поэтому снова используем `unsqueeze()`, чтобы расширить тензор, вставив новую пустую ось перед имеющимися размерностями. Затем можно передать его через модель, как обычно, что дает прогнозируемый тензор. Как и пржде, используем `torch.argmax()`, чтобы найти элемент тензора с наибольшим значением и использовать его для индексации в массиве `Cat fishClasses`. Наконец, возвращаем ответ JSON с именем класса и URL-адресом изображения, для которого выполнили прогноз.

Если вы поэкспериментируете с сервером на этом этапе, то производительность классификации может разочаровать. Разве мы не потратили уйму времени на его обучение? Да, все так, однако при повторном создании модели мы просто сделали набор слоев со стандартной инициализацией PyTorch! Поэтому неудивительно, что все так печально. Давайте добавим `load_model()` для загрузки параметров.

Возвращается только спрогнозированный класс, а не полный набор прогноза для всех классов. Конечно, можно было бы вернуть и тензор прогнозирования, но учтите, что полные тензорные выходные данные упрощают хакерам создание реплики вашей модели за счет большей *утечки информации*.

Рис. 8.2. Прогноз CATFISH

Настройка параметров модели

В главе 2 мы говорили о двух способах сохранения модели после обучения: либо путем записи всей модели на диск с помощью `torch.save()`, либо путем сохранения `state_dict()` всех весов и смещений модели (но не структуры). Для рабочей среды нужно загрузить уже обученную модель. Что же выбрать?

На мой взгляд, лучше выбрать `state_dict`. Сохранение всей модели кажется весьма заманчивым, но любые изменения в структуре модели или даже в структуре каталогов набора данных для обучения сразу станут заметны. Это может привести к проблемам с загрузкой в сторонний отдельный сервис. При переходе на другой макет не хотелось бы переделывать все заново. Также лучше избегать жесткого кодирования имени файла, сохраненного `state_dicts()`, чтобы отделить обновления модели от сервиса. Это означает, что можно перезапустить сервис с новой моделью или легко вернуться к более ранней модели.

Мы передаем имя файла как параметр, но на что он должен указывать? Предположим, что можно задать переменную окружения с именем `CATFISH_MODEL_LOCATION` и использовать ее в `load_model()`:

```
def load_model():
  m = CatfishModel()
  location = os.environ["CATFISH_MODEL_LOCATION"]
  m.load_state_dict(torch.load(location))
  return m
```

Теперь скопируйте в каталог один из файлов сохраненных весов модели (см. главу 4), `CATFISH_MODEL_LOCATION` должен указывать на этот файл.

```
export CATFISH_MODEL_LOCATION=catfishweights.pt
```

Перезапустите сервер, и вы увидите, что теперь он намного точнее!

У нас есть работающий минимальный веб-сервис (вы, скорее всего, захотите, чтобы ошибки обрабатывались более тщательно, но пускай это будет вашей самостоятельной работой!). Как запустить сервер, например, в AWS или Google Cloud? Или на чужом ноутбуке? В конце концов, мы установили несколько библиотек, чтобы все работало. Можно использовать Docker для упаковки в один *контейнер*, который можно установить в любой среде Linux (или Windows, с новой подсистемой Windows для Linux) за считаные секунды.

Сборка контейнера Docker

За последние несколько лет Docker стал стандартом упаковки приложений. Docker лежит в основе современных кластерных сред, таких как Kubernetes, используется для развертывания приложений (как вы узнаете позже из этой главы) и даже пользуется успехом на предприятиях.

Если вы раньше не сталкивались с Docker, то вот краткая справка: это программное обеспечение основано на идее доставки контейнеров. Вы указываете пакет файлов (обычно с помощью Dockerfile), Docker использует их для создания *образа*, а затем запускает этот образ в контейнере, который является изолированным процессом в вашей системе. Она может видеть только указанные файлы и программы, которые требуется запустить.

Можно поделиться Dockerfile, чтобы другие пользователи могли создавать собственные образы, но более распространена отправка созданного образа в *реестр*, представляющий собой список образов Docker, которые могут быть загружены любым пользователем, у которого есть доступ. Эти реестры могут быть открытыми или частными; корпорация Docker использует Docker Hub (https://hub.docker.com/), который является общедоступным реестром, содержащим более 100 тысяч образов Docker, но у многих компаний есть частные реестры для внутреннего использования.

Для начала нам надо написать собственный Dockerfile. Это может показаться сложным. Какую команду задать Docker для установки? Наш код? PyTorch? Conda? Python? Сам Linux? К счастью, Dockerfiles могут наследовать от других образов, поэтому мы могли бы наследовать от стандартного образа Ubuntu и установить оттуда Python, PyTorch и все остальное. Но мы можем сделать лучше!

В образах Conda есть несколько вариантов на выбор, что дает базовую установку Linux, Python и Anaconda, на которой можно выполнять сборку. Вот пример Dockerfile, который можно использовать для создания образа контейнера для нашего сервиса:

```
FROM continuumio/miniconda3:latest

ARG model_parameter_location
ARG model_parameter_name
ARG port
ARG host

ENV CATFISH_PORT=$port
ENV CATFISH_HOST=$host
ENV CATFISH_MODEL_LOCATION=/app/$model_parameter_name

RUN conda install -y flask \
  &conda install -c pytorch torchvision \
  &conda install waitress
RUN mkdir -p /app

COPY ./model.py /app
COPY ./server.py /app
COPY $model_location/$model_weights_name /app/
COPY ./run-model-service.sh /

EXPOSE $port

ENTRYPOINT ["/run-model-service.sh"]
```

Давайте разберемся, что здесь происходит. Первой строкой почти во всех файлах Docker является `FROM`, в которой указан образ Docker, от которого наследует этот файл. В данном случае это `continuumio/miniconda3:latest`. Первая часть этой строки — имя образа. Образы также имеют версии, поэтому все, что находится после двоеточия, является *тегом*, указывающим, какую версию образа мы хотим загрузить. Существует также магический тег `latest`, который мы используем для загрузки последней версии образа, который ищем. Возможно, вы захотите прикрепить свой сервис к определенной версии, чтобы не удивляться возможным последующим изменениям базового образа, которые вызывают ряд вопросов.

ARG и ENV работают с переменными. ARG указывает переменную, которая передается в Docker, когда мы создаем образ, а затем переменная может быть использована позже в Dockerfile. `ENV` позволяет вам указать переменные среды, которые будут вводиться в контейнер во время про-

гона. В нашем контейнере мы используем ARG, чтобы указать, к примеру, что порт является настраиваемым параметром, а затем используем ENV, чтобы убедиться, что конфигурация доступна нашему скрипту при запуске.

Затем RUN и COPY дают нам возможность управлять изображением, от которого мы наследовали. RUN запускает фактические команды в образе, и любые изменения сохраняются как новый *слой* образа поверх базового слоя.

COPY берет что-то из контекста сборки Docker (как правило, любые файлы из каталога, созданного командой сборки или из любых подкаталогов) и вставляет это в точку размещения в файловой системе образа. Создав / app с помощью RUN, мы затем используем COPY, чтобы переместить наш код и параметры модели в образ.

EXPOSE указывает Docker, какой порт следует открыть для внешнего мира. По умолчанию порты не открываются, поэтому мы добавляем один из них, взятый из команды ARG. Наконец, ENTRYPOINT — это стандартная команда, которая запускается при создании контейнера. Здесь мы указали скрипт, но еще не сделали его! Давайте этим займемся, прежде чем создадим образ Docker:

```
#!/bin/bash
#run-model-service.sh
cd /app
waitress-serve --call 'catfish_server:create_app'
```

Погодите, что это? Откуда взялось waitress? Дело в том, что когда мы запускали наш сервер на базе Flask, прежде чем он использовал простой веб-сервер, это предполагалось только для целей отладки. Если мы хотим запустить код в производство, нужен веб-сервер промышленного уровня. Waitress выполняет это требование. Подробности тут ни к чему, но если вам интересно, то можете ознакомиться с документацией Waitress (https://oreil.ly/x96Ir).

Со всей этой настройкой мы, наконец, можем создать образ, используя docker build:

```
docker build -t catfish-service .
```

Мы можем убедиться, что образ доступен в нашей системе, используя `docker images`:

```
>docker images
REPOSITORY                TAG                       IMAGE ID
catfish-service           latest                    e5de5ad808b6
```

Запустить сервис прогнозирования модели можно с помощью `docker run`:

```
docker run catfish-service -p 5000:5000
```

Мы также используем аргумент `-p` для сопоставления порта контейнера 5000 с портом нашего компьютера 5000. Вы должны иметь возможность вернуться к http://localhost:5000/predict, как мы делали до этого.

Возможно, вы заметили, что при локальном запуске `docker images` размер нашего образа Docker больше 4 ГБ! Это довольно много, учитывая, что мы не написали много кода. Давайте посмотрим, как уменьшить этот образ и сделать его более практичным для развертывания.

Локальное и облачное хранилище

Очевидно, что самый простой ответ на вопрос, где хранить сохраненные параметры модели, — в локальной файловой системе, либо на компьютере, либо в файловой системе в контейнере Docker. Но здесь не все так гладко. Во-первых, модель жестко закодирована в образ. Кроме того, вполне возможно, что после того, как образ собран и запущен в производство, нам нужно обновить модель. С нашим текущим Dockerfile мы должны полностью перестроить образ, даже если структура модели не изменилась. Во-вторых, большая часть размера наших образов определяется размером файла параметров. Возможно, вы не заметили, что им свойственно иметь довольно большой размер! Попробуйте сделать следующее:

```
ls -l
total 641504
-rw------- 1 ian ian 178728960 Feb  4 2018 resnet101-5d3b4d8f.pth
-rw------- 1 ian ian 241530880 Feb 18 2018 resnet152-b121ed2d.pth
-rw------- 1 ian ian 46827520  Sep 10 2017 resnet18-5c106cde.pth
-rw------- 1 ian ian 87306240  Dec 23 2017 resnet34-333f7ec4.pth
-rw------- 1 ian ian 102502400 Oct  1 2017 resnet50-19c8e357.pth
```

Если мы добавим эти модели в файловую систему при каждой сборке, образы Docker, скорее всего, будут довольно большими, что замедлит процесс отправки и извлечения данных. Предлагаю использовать локальные файловые системы или контейнеры с сопоставлением томов Docker, если вы работаете локально, но если вы выполняете развертывание в облаке, о котором мы и говорим, то есть смысл воспользоваться им в полной мере.

Файлы параметров модели можно загрузить в хранилище Azure Blob, Amazon Simple Storage Service (Amazon S3) или Google Cloud Storage и извлекать при запуске.

Можно переписать функцию `load_model()`, чтобы загрузить файл параметров при запуске:

```
from urllib.request import urlopen
from shutil import copyfileobj
from tempfile import NamedTemporaryFile

def load_model():
  m = CatfishModel()
  parameter_url = os.environ["CATFISH_MODEL_LOCATION"]
  with urlopen(url) as fsrc, NamedTemporaryFile() as fdst:
    copyfileobj(fsrc, fdst)
    m.load_state_dict(torch.load(fdst))
  return m
```

Конечно, существует много способов загрузки файлов с помощью Python; Flask даже поставляется с модулем `requests`, который легко загружает файл. Однако потенциальная проблема заключается в том, что многие средства для ее решения загружают весь файл в память перед тем, как записать его на диск. В большинстве случаев это вполне целесообразно, но при загрузке файлов параметров модели они могут попасть в диапазон гигабайт.

Поэтому в новой версии мы используем `urlopen()` и `copyfileobj() load_model()` для копирования, а `NamedTemporaryFile()` — для получения назначения, которое можно удалить в конце блока, так как к этому моменту мы уже загрузили параметры и файл больше не нужен. Это позволяет упростить Dockerfile:

```
FROM continuumio/miniconda3:latest

ARG port
ARG host

ENV CATFISH_PORT=$port
RUN conda install -y flask \
  &conda install -c pytorch torch torchvision \
  &conda install waitress
RUN mkdir -p /app

COPY ./model.py /app
COPY ./server.py /app
COPY ./run-model-service.sh /

EXPOSE $port

ENTRYPOINT ["/run-model-service.sh"]
```

Когда мы запускаем код с помощью `docker run`, то передаем переменную среды:

```
docker run catfish-service --env CATFISH_MODEL_LOCATION=[URL]
```

Теперь сервис извлекает параметры из URL-адреса, а образ Docker, скорее всего, примерно на 600–700 МБ меньше исходного.

В этом примере мы полагаем, что файл параметров модели находится в общедоступном месте. Если вы развертываете сервисную модель, то, скорее всего, вы с подобным не столкнетесь, а вместо этого будете извлекать данные из слоя облачного хранилища, например из Amazon S3, Google Cloud Storage или хранилища BLOB-объектов Azure. Вам нужно будет использовать API соответствующего провайдера для загрузки файла и получения сведений об учетной записи, чтобы получить к нему доступ. Я не буду подробно описывать здесь ни то ни другое.

Итак, у нас есть сервисная модель, способная коммуницировать через HTTP с JSON. Теперь мы должны убедиться, что можем контролировать ее, когда она делает прогнозы.

Логирование и телеметрия

Одна вещь, которой у нас нет в текущем сервисе, — это любая концепция логирования. И хотя сервис невероятно прост и, возможно, такая опция ему и не нужна (за исключением случаев обнаружения наших ошибок), она бы пригодилась в случае необходимости отслеживания того, что на самом деле прогнозируется. В какой-то момент мы захотим оценить модель; как мы можем сделать это без производственных данных?

Давайте предположим, что у нас есть метод `send_to_log()`, который принимает `dict` Python и отправляет его в другое место (возможно, в кластер Apache Kafka, который выполняет бэкап в облачное хранилище). Мы можем отправлять соответствующую информацию с помощью этого метода каждый раз, когда делаем прогноз:

```
import uuid
import logging
logging.basicConfig(level=logging.INFO)

def predict():
  img_url = request.image_url
  img_tensor = open_image(BytesIO(response.content))
  start_time = time.process_time()
  prediction = model(img_tensor)
  end_time = time.process_time()
  predicted_class = CatfishClasses[torch.argmax(prediction)]
  send_to_log(
    {"image": img_url,
    "prediction": predicted_class},
    "predict_tensor": prediction,
    "img_tensor": img_tensor,
    "predict_time": end_time-start_time,
    "uuid":uuid.uuid4()
    })
  return jsonify({"image": img_url, "prediction": predicted_class})

def send_to_log(log_line):
  logger.info(log_line)
```

Имея некоторые дополнительные возможности для вычисления времени прогнозирования, при каждом запросе этот метод отправляет сообщение с важной информацией логгеру или внешнему ресурсу, например, URL-адрес образа, прогнозируемый класс, фактический тензор прогнозирования и даже полный тензор образа на случай, если предоставленный URL-адрес является временным. Мы также включили универсально

сгенерированный уникальный идентификатор (UUID), чтобы на этот прогноз всегда можно было ссылаться позднее, если его прогнозируемый класс необходимо исправить. В реальном развертывании вы бы включили такие штуки, как `user_id`s и т. п., чтобы следующие системы цепочки могли предоставить пользователям возможность указать, был прогноз верным или нет, незаметно генерируя больше обучающих данных для дальнейших обучающих итераций модели. Теперь мы можем развернуть контейнер в облаке. Давайте кратко рассмотрим использование Kubernetes для размещения и масштабирования сервиса.

Развертывание в Kubernetes

Не будем слишком углубляться в Kubernetes, поэтому лишь кратко рассмотрим основы, в том числе и то, как быстро запустить сервис.[1] *Kubernetes* (также известный как *k8s*) быстро становится основным кластерным фреймворком в облаке. Это программное обеспечение было создано на основе оригинального программного обеспечения Google для управления кластерами Borg и содержит все части и связующие для формирования отказоустойчивого и надежного способа для запуска сервиса, включая балансировку нагрузки, квоты на ресурсы, масштабирование, управление трафиком, совместное использование секретов и многое другое.

Вы можете загрузить и настроить Kubernetes на локальном компьютере или в учетной записи в облаке; я советую использовать хостинговый сервис, где управление самим Kubernetes осуществляется облачным провайдером и вам остается только управлять им. Для развертывания мы используем сервис Google Kubernetes Engine (GKE), но вы также можете развернуть его в Amazon, Azure или DigitalOcean.

[1] Книга *Cloud Native DevOps with Kubernetes*, написанная Джоном Арунделом и Джастином Домингусом (O'Reilly), позволит глубже погрузиться в эту концепцию. **На русском языке:** Арундел Джон, Домингус Джастин. Kubernetes для DevOps: развертывание, запуск и масштабирование в облаке. — СПб.: Питер, 2020. — 384 с. (Серия «Бестселлеры O'Reilly»).

Установка на Google Kubernetes Engine

Чтобы использовать GKE, вам нужен аккаунт Google Cloud (https://cloud.google.com/). Кроме того, запуск сервиса в GKE не бесплатный. С другой стороны, если вы являетесь новым пользователем в Google Cloud, то получите бесплатный кредит в размере 300 долларов. Мы не собираемся сжигать больше чем доллар или два.

Получив учетную запись, загрузите для своей системы `gcloud SDK` (https://cloud.google.com/sdk). После установки мы можем использовать его, чтобы установить приложение `kubectl`. Оно понадобится для взаимодействия с кластером Kubernetes, который мы будем создавать:

```
gcloud login
gcloud components install kubectl
```

Затем нам нужно создать *новый проект*, Google Cloud организует вычислительные ресурсы в вашем аккаунте следующим образом:

```
gcloud projects create ml-k8s --set-as-default
```

Затем мы пересобираем образ Docker и отмечаем его так, чтобы его можно было передать во внутренний реестр, который предоставляет Google (используйте `gcloud` для аутентификации), а затем мы можем использовать `docker push` для отправки образа контейнера в облако. Обратите внимание, что мы также отмечаем сервис тегом версии `v1`, чего мы не делали до этого:

```
docker build -t gcr.io/ml-k8s/catfish-service:v1 .
gcloud auth configure-docker
docker push gcr.io/ml-k8s/catfish-service:v1
```

Создание кластера k8s

Теперь мы можем создать кластер Kubernetes. В следующей команде мы создаем один кластер с двумя узлами n1-standard-1, самыми дешевыми и экономичными экземплярами Google. Если вы сильно экономите, то можете создать кластер только с одним узлом.

```
gcloud container clusters create ml-cluster --num-nodes=2
```

Для полной инициализации нового кластера может потребоваться несколько минут. Когда все будет готово, можем использовать `kubectl` для развертывания нашего приложения!

```
kubectl run catfish-service
--image=gcr.io/ml-k8s/catfish-service:v1
--port 5000
--env CATFISH_MODEL_LOCATION=[URL]
```

Обратите внимание, что здесь мы передаем расположение файла параметра модели в качестве параметра среды, как мы это делали с командой `docker run` на локальной машине. С помощью `kubectl get pods` вы сможете увидеть, какие модули работают в кластере. *Под* (*pod*) — это группа из одного или нескольких контейнеров, включающая спецификацию о том, как их запускать и управлять ими. Для наших целей мы запускаем модель в одном контейнере в одном поде. Вот что вы должны увидеть:

```
NAME                                READY  STATUS   RESTARTS  AGE
gcr.io/ml-k8s/catfish-service:v1    1/1    Running  0 4m15s
```

Итак, теперь мы можем видеть, что наше приложение работает, но как нам с ним коммуницировать? Для этого нам нужно развернуть *сервис*, в данном случае балансировщик нагрузки, который сопоставляет внешний IP-адрес с нашим внутренним кластером:

```
kubectl expose deployment catfish-service
--type=LoadBalancer
--port 80
--target-port 5000
```

Затем вы можете посмотреть на запущенные сервисы, используя `kubectl get services` для получения внешнего IP:

```
kubectl get service

NAME              CLUSTER-IP     EXTERNAL-IP     PORT(S)         AGE
catfish-service   10.3.251.122   203.0.113.0     80:30877/TCP    3d
```

Теперь должна появиться возможность перейти на http://external-ip/predict на вашем локальном компьютере. Ура! Мы также можем проверить логи пода без логирования:

```
kubectl logs catfish-service-xxdsd
>> log response
```

Теперь у нас есть работающее развертывание в кластере Kubernetes. Давайте рассмотрим некоторые из возможностей, которые оно дает.

Сервисы масштабирования

Допустим, мы решили, что одного пода для обработки всего трафика, поступающего в сервис прогнозирования, недостаточно. В традиционном развертывании нужно было бы задействовать новые серверы, добавить их в балансировщики нагрузки и решить, что делать в случае сбоя одного из серверов. Но Kubernetes делает все гораздо проще. Давайте удостоверимся, что все три копии сервиса работают:

```
kubectl scale deployment hello-web --replicas=3
```

Если вы продолжите смотреть на `kubectl get pods`, то вскоре увидите, что Kubernetes выводит еще два пода из образа Docker и подключает их к балансировщику нагрузки. Давайте еще посмотрим, что произойдет, если мы удалим один из подов:

```
kubectl delete pod [PODNAME]
kubectl get pods
```

Вы увидите, что указанный под был удален. Но кроме этого появился новый под на замену! Мы указали Kubernetes, что нужно запустить три копии образа, и поскольку мы удалили одну, кластер запускает новый под, чтобы убедиться, что количество реплик соответствует запрашиваемому. То же самое относится к обновлению приложения, поэтому давайте рассмотрим и его.

Обновления и очистка

Когда дело доходит до обновления сервисного кода, создается новая версия контейнера с тегом `v2`:

```
docker build -t gcr.io/ml-k8s/catfish-service:v2 .
docker push gcr.io/ml-k8s/catfish-service:v2
```

Далее кластеру дается команда использовать новый образ для развертывания:

```
kubectl set image deployment/catfish-service
  catfish-service=gcr.io/ml-k8s/catfish-service:v2
```

Продолжайте наблюдать через `kubectl get pods`, и вы увидите, что развертываются новые поды с новыми образами, а поды со старыми образами удаляются. Kubernetes автоматически позаботится о соединениях и удалении старых контейнеров из балансировщика нагрузки.

Наконец, если вы закончили свои эксперименты с кластером, вам следует навести порядок, чтобы избежать дополнительных неожиданных сюрпризов:

```
kubectl delete service catfish-service
gcloud container clusters delete ml-k8s
```

На этом мы завершаем наш мини-тур по Kubernetes; теперь у вас достаточно информации для работы, рекомендую вам обязательно посетить вебсайт Kubernetes для получения дополнительной информации о системе (и поверьте мне, ее много!).

Мы рассмотрели способы, с помощью которых можно развернуть код на основе Python, но, что удивительно, PyTorch не ограничивается только Python. В следующем разделе вы увидите, как TorchScript раскрывает перед нами более широкий мир C ++, а также познакомитесь с некоторыми оптимизациями для наших обычных моделей Python.

TorchScript

Если вы помните из введения (знаю, это было давным-давно!), основное различие между PyTorch и TensorFlow состоит в том, что у TensorFlow есть представление модели на основе графа, тогда как PyTorch реализует метод автоматического дифференцирования на основе ленты. «Энергичный» метод позволяет вам применять всевозможные динамические подходы к определению и обучению моделей, что делает PyTorch привлекательным для исследовательских целей.

С другой стороны, представление на основе графа может быть статичным, но его питает стабильность; применять оптимизацию к представлению графа можно, только будучи уверенными в том, что ничего не поменяется.

И так же как TensorFlow перешел на энергичное выполнение в версии 2.0, версия 1.0 PyTorch представила TorchScript, который позволяет нам использовать преимущества систем на основе графов и не отказываться полностью от гибкости PyTorch. Для этого есть два способа, которые можно смешивать: трассировка и использование TorchScript напрямую.

Трассировка

PyTorch 1.0 идет в комплекте с механизмом трассировки JIT, который трансформирует существующий модуль или функцию PyTorch в такой же для TorchScript. Он передает пример тензора через модуль и возвращает результат `ScriptModule`, который содержит представление исходного кода TorchScript.

Давайте посмотрим на трассировку AlexNet:

```
model = torchvision.models.AlexNet()
traced_model = torch.jit.trace(model,
                torch.rand(1, 3, 224, 224))
```

Теперь все *заработает*, но вы получите сообщение от интерпретатора Python, которое вынудит сделать паузу:

```
TracerWarning: Trace had nondeterministic nodes. Nodes:
%input.15 :
Float(1, 9216) = aten::dropout(%input.14, %174, %175),
scope: AlexNet/Sequential[classifier]/Dropout[0]
%input.18 :
Float(1, 4096) = aten::dropout(%input.17, %184, %185),
scope: AlexNet/Sequential[classifier]/Dropout[3]

This may cause errors in trace checking.
To disable trace checking, pass check_trace=False to torch.jit.trace()

_check_trace([example_inputs], func, executor_options,
module, check_tolerance, _force_outplace)
/home/ian/anaconda3/lib/
python3.6/site-packages/torch/jit/__init__.py:642:
TracerWarning: Output nr 1. of the traced function does not
match the corresponding output of the Python function. Detailed error:

Not within tolerance rtol=1e-05 atol=1e-05 at input[0, 22]
(0.010976361110806465 vs. -0.005604125093668699)
and 996 other locations (99.00%)
_check_trace([example_inputs], func,
executor_options, module, check_tolerance
_force_outplace)
```

Что тут происходит? Когда мы создаем AlexNet (или другие модели), она создается в режиме *обучения*. Во время обучения во многих моделях, таких как AlexNet, мы используем слой `Dropout`, который случайным образом убивает активации, когда тензор проходит через сеть. JIT дважды отправляет случайный тензор, который мы сгенерировали с помощью модели, сравнивает их и отмечает, что слои `Dropout` не совпадают. Поэтому вам нужно быть внимательными при трассировке; она не может справиться с не повторяемостью или потоком команд управления. Если ваша модель использует эти функции, вам придется использовать TorchScript напрямую, по крайней мере для части преобразования.

Однако в случае с AlexNet исправить это просто: с помощью `model.eval()` мы переключим модель в режим оценки. Если вы снова запустите линию трассировки, то обнаружите, что она завершается нормально. Мы также можем задать `print()` для трассируемой модели, чтобы увидеть, из чего она состоит:

```
print(traced_model)

TracedModule[AlexNet](
(features): TracedModule[Sequential](
  (0): TracedModule[Conv2d]()
  (1): TracedModule[ReLU]()
  (2): TracedModule[MaxPool2d]()
  (3): TracedModule[Conv2d]()
  (4): TracedModule[ReLU]()
  (5): TracedModule[MaxPool2d]()
  (6): TracedModule[Conv2d]()
  (7): TracedModule[ReLU]()
  (8): TracedModule[Conv2d]()
  (9): TracedModule[ReLU]()
  (10): TracedModule[Conv2d]()
  (11): TracedModule[ReLU]()
  (12): TracedModule[MaxPool2d]()
)
(classifier): TracedModule[Sequential](
  (0): TracedModule[Dropout]()
  (1): TracedModule[Linear]()
  (2): TracedModule[ReLU]()
  (3): TracedModule[Dropout]()
  (4): TracedModule[Linear]()
  (5): TracedModule[ReLU]()
  (6): TracedModule[Linear]()
 )
)
```

Мы также можем увидеть код, созданный JIT, если вызовем `print(traced_model.code)`:

```
def forward(self,
  input_1: Tensor) -> Tensor:
  input_2 = torch._convolution(input_1, getattr(self.features, "0").weight,
  getattr(self.features, "0").bias,
  [4, 4], [2, 2], [1, 1], False, [0, 0], 1, False, False, True)
  input_3 = torch.threshold_(input_2, 0., 0.)
  input_4, _0 = torch.max_pool2d_with_indices
  (input_3, [3, 3], [2, 2], [0, 0], [1, 1], False)
  input_5 = torch._convolution(input_4, getattr
  (self.features, "3").weight, getattr(self.features, "3").bias,
  [1, 1], [2, 2], [1, 1], False, [0, 0], 1, False, False, True)
  input_6 = torch.threshold_(input_5, 0., 0.)
  input_7, _1 = torch.max_pool2d_with_indices
  (input_6, [3, 3], [2, 2], [0, 0], [1, 1], False)
  input_8 = torch._convolution(input_7, getattr(self.features, "6").weight,
  getattr
  (self.features, "6").bias,
  [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  input_9 = torch.threshold_(input_8, 0., 0.)
  input_10 = torch._convolution(input_9, getattr
  (self.features, "8").weight, getattr(self.features, "8").bias,
  [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  input_11 = torch.threshold_(input_10, 0., 0.)
  input_12 = torch._convolution(input_11, getattr
  (self.features, "10").weight, getattr(self.features, "10").bias,
  [1, 1], [1, 1], [1, 1], False, [0, 0], 1, False, False, True)
  input_13 = torch.threshold_(input_12, 0., 0.)
  x, _2 = torch.max_pool2d_with_indices
  (input_13, [3, 3], [2, 2], [0, 0], [1, 1], False)
  _3 = ops.prim.NumToTensor(torch.size(x, 0))
  input_14 = torch.view(x, [int(_3), 9216])
  input_15 = torch.dropout(input_14, 0.5, False)
  _4 = torch.t(getattr(self.classifier, "1").weight)
  input_16 = torch.addmm(getattr(self.classifier, "1").bias,
    input_15, _4, beta=1, alpha=1)
  input_17 = torch.threshold_(input_16, 0., 0.)
  input_18 = torch.dropout(input_17, 0.5, False)
  _5 = torch.t(getattr(self.classifier, "4").weight)
  input_19 = torch.addmm(getattr(self.classifier, "4").bias,
    input_18, _5, beta=1, alpha=1)
  input = torch.threshold_(input_19, 0., 0.)
  _6 = torch.t(getattr(self.classifier, "6").weight)
  _7 = torch.addmm(getattr(self.classifier, "6").bias, input,
    _6, beta=1, alpha=1)
  return _7
```

Затем модель (код и параметры) можно сохранить с помощью `torch.jit.save`:

```
torch.jit.save(traced_model, "traced_model")
```

Вот так работает трассировка. Давайте теперь посмотрим, как использовать TorchScript.

Выполнение скриптов

Вас может удивить, почему нельзя просто выполнить трассировку всего. Несмотря на то что трассировщик прекрасно справляется со своими задачами, у него есть ограничения. Например, невозможно оттрассировать подобную простую функцию за один проход:

```
import torch

def example(x, y):
  if x.min() > y.min():
      r = x
  else:
      r = y
  return r
```

Одна трассировка через функцию проведет нас по одному пути, то есть функция не будет конвертирована правильно. В этих случаях мы можем использовать TorchScript, который является ограниченным подмножеством Python, и создать скомпилированный код. Мы используем *аннотацию*, чтобы сообщить PyTorch об использовании TorchScript, поэтому реализация TorchScript будет выглядеть следующим образом:

```
@torch.jit.script
def example(x, y):
    if x.min() > y.min():
        r = x
    else:
        r = y
    return r
```

К счастью, в этой функции мы не используем отсутствующие в TorchScript конструкции и не ссылаемся на какое-либо глобальное состояние, поэтому все будет работать. Если бы мы создавали новую ар-

хитектуру, нужно было бы наследовать от `torch.jit.ScriptModule` вместо `nn.Module`. А как можно использовать другие модули (например, слои на базе CNN), если все модули должны наследовать от другого класса? Есть отличие, не правда ли? Дело в том, что мы можем смешивать и сопоставлять и то и другое, используя TorchScript и трассируемые объекты по желанию. Давайте вернемся к нашей структуре CNNNet/AlexNet из главы 3 и посмотрим, как можно преобразовать ее в TorchScript с помощью комбинации этих методов. Для краткости реализуем только компонент функций:

```
class FeaturesCNNNet(torch.jit.ScriptModule):
    def __init__(self, num_classes=2):
        super(FeaturesCNNNet, self).__init__()
        self.features = torch.jit.trace(nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1), nn.ReLU(),
            nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2)
        ), torch.rand(1,3,224,224))

    @torch.jit.script_method
    def forward(self, x):
        x = self.features(x)
        return x
```

Отметим две вещи. Во-первых, внутри классов мы аннотируем с помощью `@torch.jit.script_method`. Во-вторых, хотя возможно трассировать каждый слой по отдельности, мы воспользовались оберткой `nn.Sequential`, чтобы запустить трассировку. Попробуйте реализовать блок `classifier` самостоятельно, чтобы понять, как он работает. Помните, что вместо обучения нужно переключить слои `Dropout` в режим `eval()`, и ваш входной тензор трассировки должен иметь форму `[1, 256, 6, 6]`, что обусловлено понижающей дискретизацией, которую выполняет блок `features`. И да, можно сохранить эту сеть с помощью `torch.jit.save`, как мы это делали для трассируемого модуля. Давайте посмотрим на возможности и ограничения TorchScript.

Ограничения TorchScript

По сравнению с Python самое большое ограничение в TorchScript, по крайней мере на мой взгляд, заключается в небольшом количестве доступных типов. В табл. 8.1 показаны доступные и недоступные типы.

Таблица 8.1. Доступные типы Python в TorchScript

Тип	Описание
`tensor`	Тензор PyTorch любого типа, размера или бэкенда
`tuple[T0, T1,…]`	Кортеж, содержащий подтипы T0, T1 и т. д. (например, `tuple[tensor, tensor]`)
`boolean`	Boolean
`str`	String
`int`	Int
`float`	Float
`list`	Список типов T
`optional[T]`	Либо None, либо тип T
`dict[K, V]`	`dict` с ключами типа K и значениями типа V; K может быть только `str`, int или float

Еще одна опция, которую вы не можете сделать на стандартном Python, — это функция, которая смешивает типы возвращаемых значений. В TorchScript следующее недопустимо:

```
def maybe_a_string_or_int(x):
  if x > 3:
    return "bigger than 3!"
  else
    return 2
```

Конечно, в Python это тоже не очень хорошая идея, но динамический контроль типов языка позволит это сделать. TorchScript статически типирован (что помогает в применении оптимизаций), поэтому в аннотированном коде TorchScript так делать не получится. Кроме того, TorchScript предполагает, что каждый параметр, передаваемый в функцию, является тензором, а это может вызвать некоторое недоумение, если вы не понимаете, что происходит:

```
@torch.jit.script
def add_int(x,y):
  return x + y

print(add_int.code)
>> def forward(self,
  x: Tensor,
  y: Tensor) -> Tensor:
  return torch.add(x, y, alpha=1)
```

Чтобы использовать разные типы, потребуются декораторы типов Python 3:

```
@torch.jit.script
def add_int(x: int, y: int) -> int:
  return x + y
print(add_int.code)
>> def forward(self,
  x: int,
  y: int) -> int:
return torch.add(x, y)
```

Как вы видите, классы поддерживаются, но есть несколько ошибок. Все методы в классе должны соответствовать TorchScript, но хотя этот код и выглядит правильно, работать он не будет:

```
@torch.jit.script
class BadClass:
  def __init__(self, x)
    self.x = x

  def set_y(y)
    self.y = y
```

Это следствие статической типизации TorchScript. Все переменные экземпляра должны быть объявлены во время `__init__` и не могут быть представлены где-либо еще. Да, и даже не думайте включать в класс какие-либо выражения, которых нет в методе, — в TorchScript они запрещены.

Полезная функция TorchScript, являющегося подмножеством Python, заключается в том, что перевод можно выполнять по частям, а промежуточный код все еще действует и выполняется на Python. Код, совместимый с TorchScript, может приводить к появлению несовместимого кода, и хотя вы не сможете выполнять `torch.jit.save()` до тех пор, пока не преобразуете весь несовместимый код, вы все равно сможете запускать все на Python.

Узнать больше вы можете в документации PyTorch (https://oreil.ly/sS0o7), которая подробно описывает такие вещи, как область видимости (в основном это стандартные правила Python), однако той информации, которая представлена в этой книге, вполне достаточно для преобразования всех описанных мной моделей. Я не буду отсылать вас к разным источникам, а предложу рассмотреть реализацию одной из моделей TorchScript на C++.

Работа с libTorch

В дополнение к TorchScript PyTorch 1.0 представил `libTorch`, библиотеку языка C++ для взаимодействия с PyTorch. Доступны различные уровни взаимодействия C++. Низшие уровни — `ATen` и `autograd`, реализация тензора и автоматического дифференцирования в C++, на которых построен сам PyTorch. Кроме того, имеется пользовательский интерфейс на C++, который дублирует питонский API PyTorch на C++, интерфейс к TorchScript и, наконец, интерфейс расширения, который позволяет определять новые пользовательские операторы C++/CUDA и осуществлять реализацию PyTorch на Python. В этой книге мы рассматриваем только пользовательский интерфейс C++ и интерфейс TorchScript, дополнительную информацию смотрите в документации PyTorch. Давайте начнем с `libTorch`.

Получение libTorch и Hello World

Нам необходим компилятор C++ и способ построения программ C++ на компьютере. Это одна из немногих частей книги, где Google Colab не подойдет; возможно, потребуется создать виртуальную машину в Google Cloud, AWS или Azure, если у вас нет легкого доступа к окну терминала. (Бьюсь об заклад, все, кто проигнорировал мой совет не создавать выделенную машину, сейчас невероятно гордятся собой!) Для `libTorch` потребуется компилятор C++ и *CMake*, поэтому давайте установим их. В системе на базе Debian используйте эту команду:

```
apt install cmake g++
```

Если у вас система на базе Red Hat, используйте эту команду:

```
yum install cmake g++
```

Далее нужно скачать библиотеку libTorch. В целях упрощения процесса будем использовать дистрибутив libTorch на базе CPU, а не работать с дополнительными зависимостями CUDA, которые дает дистрибутив на базе GPU. Создайте каталог torchscript_export и получите дистрибутив:

```
wget https://download.pytorch.org/libtorch/cpu/
             libtorch-shared-with-deps-latest.zip
```

Используйте unzip, чтобы распаковать ZIP-файл (появится новый каталог libtorch), и создайте каталог helloworld. Сюда мы добавим минимальный *CMakeL ists.txt*, который *CMake* будет использовать для компиляции исполняемой программы:

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(helloworld)

find_package(Torch REQUIRED)

add_executable(helloworld helloworld.cpp)
target_link_libraries(helloworld "${TORCH_LIBRARIES}")
set_property(TARGET helloword PROPERTY CXX_STANDARD 11)
```

И тогда *helloworld.cpp* выглядит следующим образом:

```
#include <torch/torch.h>
#include <iostream>

int main() {
  torch::Tensor tensor = torch::ones({2, 2});
  std::cout << tensor << std::endl;
}
```

Создайте каталог *build* и запустите cmake, убедившись, что мы обеспечили *абсолютный* путь к дистрибутиву libtorch:

```
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=/absolute/path/to/libtorch ..
cd ..
```

Теперь можно запустить простой make, чтобы создать исполняемый файл:

```
make
./helloworld
```

```
1  1
1  1
[ Variable[CPUType]{2,2} ]
```

Поздравляю с созданием первой программы на C++ с помощью `libTorch`! Теперь давайте расширим ее и посмотрим, как использовать библиотеку для загрузки в модель, которую мы сохранили с помощью `torch.jit.save()`.

Импорт модели TorchScript

Мы собираемся экспортировать нашу полную модель CNNNet, о которой шла речь в главе 3, и загрузить ее в C++. В Python создайте экземпляр CNNNet, переключите его в режим `eval()`, чтобы игнорировать `Dropout`, выполнять трассировку и сохранять на диск:

```
cnn_model = CNNNet()
cnn_model.eval()
cnn_traced = torch.jit.trace(cnn_model, torch.rand([1,3,224,224]))
torch.jit.save(cnn_traced, "cnnnet")
```

В C++ создайте новый каталог с именем *load-cnn* и добавьте новый файл *CMakeLists.txt* file:

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(load-cnn)

find_package(Torch REQUIRED)

add_executable(load-cnn.cpp load-cnn.cpp)
target_link_libraries(load-cnn "${TORCH_LIBRARIES}")
set_property(TARGET load-cnn PROPERTY CXX_STANDARD 11)
```

Теперь создадим программу C++, `load-cnn.cpp`:

```
#include <torch/script.h>
#include <iostream>
#include <memory>

int main(int argc, const char* argv[]) {

  std::shared_ptr<torch::jit::script::Module> module =
                 torch::jit::load("cnnnet");

  assert(module != nullptr);
  std::cout << "model loaded ok\n";
```

```
  // Create a vector of inputs.
  std::vector<torch::jit::IValue> inputs;
  inputs.push_back(torch::rand({1, 3, 224, 224}));

  at::Tensor output = module->forward(inputs).toTensor();

  std::cout << output << '\n'
}
```

В этой небольшой программе есть кое-что новенькое, хотя в основном все должно напоминать вам Python PyTorch API. Первое, что мы делаем, — загружаем модель TorchScript с помощью `torch::jit::load` (в Python с помощью `torch.jit.load`). Делаем проверку нулевого указателя, чтобы убедиться, что модель загружена правильно, а затем тестируем модель со случайным тензором. Хотя это довольно легко сделать с помощью `torch::rand`, при взаимодействии с моделью TorchScript нужно создать вектор входных данных `torch::jit::IValue`, а не просто обычный тензор, что обусловлено способом реализации TorchScript в C++. Как только все будет сделано, можно передать тензор через загруженную модель и, наконец, обратно записать результат в стандартный вывод. Компилируем так же, как и предыдущую программу:

```
mkdir build
cd build
cmake -DCMAKE_PREFIX_PATH=/absolute/path/to/libtorch ..
cd ..
make
./load-cnn

0.1775
0.9096
[ Variable[CPUType]{2} ]
```

И вуаля! Программа на C++, которая выполняет пользовательскую модель без особых усилий с нашей стороны, готова. Имейте в виду, что интерфейс C++ еще находится в стадии бета-версии, поэтому некоторые детали могут измениться. Обязательно ознакомьтесь с документацией, прежде чем приступать к работе!

Заключение

Надеюсь, теперь вы понимаете, как взять обученную (и отлаженную) модель и превратить ее в веб-сервис Docker, который можно развернуть через Kubernetes. Вы также узнали, как использовать функции JIT и TorchScript для оптимизации моделей и как загружать модели TorchScript в C++, что обеспечивает низкоуровневую интеграцию нейронных сетей, так же как в Python.

Очевидно, что мы не можем охватить все, что связано с промышленным использованием и обслуживанием моделей, в одной главе. Мы описали развертывание сервиса, но это еще не конец; существует также постоянный мониторинг точности переобучения и тестирования по базовым показателям сервиса, кроме того, есть более сложные схемы контроля версий, чем те, которые я здесь описал. Рекомендую заносить в лог как можно больше информации и использовать ее для обучения, а также для мониторинга.

Что касается TorchScript, это еще только начало, но уже начинают появляться некоторые привязки для других языков (например, Go и Rust); к 2020 году подключить модель PyTorch к любому популярному языку станет совсем легко.

Я намеренно не стал описывать информацию, которая не совсем соответствует теме этой книги. Еще во введении я обещал, что вы сможете производить все операции, описанные в книге, используя один GPU, поэтому мы не затрагивали тему поддержки PyTorch для распределенного обучения и вывода. Кроме того, если вы будете читать об экспорте модели PyTorch, то почти наверняка встретите множество ссылок на Open Neural Network Exchange (ONNX). Этот метод, разработанный совместными усилиями компаний Microsoft и Facebook, был основным стандартным методом экспорта моделей до появления TorchScript. Модели можно экспортировать в TorchScript с помощью метода трассировки, а затем импортировать в другие среды, такие как Caffe2, Microsoft Cognitive Toolkit и MXNet. ONNX по-прежнему поддерживается и активно работает в PyTorch v1.x, но, по всей видимости, для экспорта моделей предпочтительным способом является TorchScript. Изучите раздел «Дополнительные источники» для более подробной информации об ONNX.

Успешно создав, отладив и развернув модели, мы переходим к последней главе, в которой рассмотрим, как некоторые компании используют PyTorch.

Дополнительные источники

- Документация Flask, http://flask.pocoo.org/
- Документация Waitress, https://oreil.ly/bnelI
- Документация Docker, https://docs.docker.com/
- Документация Kubernetes (k8s), https://oreil.ly/jMVcN
- Документация TorchScript, https://oreil.ly/sS0o7
- Open Neural Network Exchange, https://onnx.ai/
- Использование ONNX с PyTorch, https://oreil.ly/UXz5S
- Распределенное обучение с PyTorch, https://oreil.ly/Q-Jao

ГЛАВА 9

PyTorch на практике

В последней главе мы рассмотрим, как разные люди и компании используют PyTorch. Вы также узнаете о нескольких новых технологиях, в том числе о технологии изменения размера изображений, создания текста и изображений, которые могут обмануть нейронные сети. Сосредоточимся на том, как начать работу с существующими библиотеками, а не с нуля. Надеюсь, что книга станет для вас трамплином в дальнейших исследованиях.

Начнем с обзора некоторых последних подходов, позволяющих выжать максимум из данных.

Аугментация данных: смешанная и сглаженная

Еще в главе 4 мы рассматривали различные способы аугментации данных с целью снижения переобучения модели. Возможность делать больше при меньшем объеме данных естественно вызывает повышенный интерес в сфере исследований глубокого обучения, и в этом разделе мы рассмотрим два набирающих популярность способа выжать из данных максимум. На примере этих подходов вы увидите, как изменять способ вычисления функции потерь, так что это будет хорошим испытанием для более гибкого цикла обучения, который мы только что создали.

mixup

mixup — это интересная техника аугментации, представляющая собой нестандартные ожидания от модели. Наше нормальное понимание модели состоит в том, что мы отправляем ей изображение, например, как на рис. 9.1, и хотим, чтобы модель вернула результат: на изображении представлена лиса.

Но, как вы знаете, мы не получаем результат от модели; мы получаем тензор из всех возможных классов, а элементом этого тензора с наибольшим значением является класс *лиса*. В идеальном сценарии у нас был бы тензор, равный 0, за исключением 1 в классе лисы.

Однако нейронной сети трудно это сделать! Всегда будет неопределенность, а функции активации, такие как `softmax`, затрудняют переход тензоров к 1 или 0. mixup использует это, задавая вопрос: «К какому классу относится рисунок 9.2?»

Рис. 9.1. Лиса

Рис. 9.2. Смесь кота и лисы

Может показаться немного странным, но на картинке — на 60 % кошка и на 40 % лиса. Что, если вместо того, чтобы заставлять модель делать окончательное предположение, мы могли бы сделать ее ориентированной на два класса? То есть это означало бы, что при обучении выходной тензор не столкнется с проблемой приближения к 1 и при этом невозможности ее достичь, и мы могли бы изменять каждое смешанное изображение в разной степени, улучшая способность модели к обобщению.

Но как рассчитать функцию потерь этого смешанного изображения? Итак, если p — это процент первого изображения в смешанном изображении, то мы имеем простую линейную комбинацию:

```
p * loss(image1) + (1-p) * loss(image2)
```

Она должна прогнозировать эти изображения, верно? И нам нужно масштабировать в соответствии с количеством изображений в конечном смешанном изображении, поэтому эта новая функция потерь кажется логичной. Чтобы выбрать p, мы могли бы просто использовать случайные числа из нормального или равномерного распределения, как мы сделали бы во многих других случаях. Однако авторы работ по mixup определили, что образцы из бета-*распределения* на практике работают намного лучше.[1] Не знаете, как выглядит бета-распределение? Я тоже не знал, пока не увидел эту работу! На рис. 9.3 показано бета-распределение с учетом описанных в работе характеристик.

U-образная форма сообщает о том, что в большинстве случаев наше смешанное изображение будет либо тем, либо другим. Это очевидно, так как мы можем предположить, что сети будет сложнее разрабатывать mixup 50/50, чем 90/10.

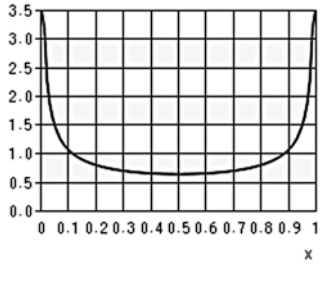

Рис. 9.3. Бета-распределение, где α = β

[1] См. «mixup: Beyond Empirical Risk Minimization», Hongyi Zhang et al. (2017).

Ниже вы видите модифицированный цикл обучения, который принимает новый дополнительный загрузчик данных, `mix_loader`, и смешивает пакеты:

```
def train(model, optimizer, loss_fn, train_loader, val_loader,
epochs=20, device, mix_loader):
  for epoch in range(epochs):
    model.train()
    for batch in zip(train_loader,mix_loader):
      ((inputs, targets),(inputs_mix, targets_mix)) = batch
      optimizer.zero_grad()
      inputs = inputs.to(device)
      targets = targets.to(device)
      inputs_mix = inputs_mix.to(device)
      target_mix = targets_mix.to(device)

      distribution = torch.distributions.beta.Beta(0.5,0.5)
      beta = distribution.expand(torch.zeros(
             batch_size).shape).sample().to(device)

      # Нам нужно преобразовать форму бета,
      # чтобы она была в тех же размерностях, что и наш входной тензор
      # [batch_size, channels, height, width]

      mixup = beta[:, None, None, None]

      inputs_mixed = (mixup * inputs) + (1-mixup * inputs_mix)

      # Targets are mixed using beta as they have the same shape

      targets_mixed = (beta * targets) + (1-beta * inputs_mix)

      output_mixed = model(inputs_mixed)

      # Умножьте потери на бета и 1-бета,
      # сложите и получите среднее значение из двух смешанных потерь

      loss = (loss_fn(output, targets) * beta
             + loss_fn(output, targets_mixed)
             * (1-beta)).mean()

      # Метод обучения, в обычном режиме, здесь и далее

      loss.backward()
      optimizer.step()
      …
```

После получения двух пакетов мы используем `torch.distribu tion.Beta`, чтобы сгенерировать серию смешанных параметров с помощью метода

expand для получения тензора [1, batch_size]. Мы могли бы итерировать пакет и генерировать параметры по очереди, но так получается аккуратнее, и не забывайте, что графические процессоры любят матричное умножение, так что в итоге он будет быстрее выполнять все вычисления пакета сразу (об этом говорится в главе 7, исправление преобразования BadRandom). Сначала мы умножаем весь пакет на этот тензор, а затем умножаем пакет, который смешиваем на 1 - mix_factor_tensor, используя транслирование (см. главу 1).

Затем мы берем потери прогнозов относительно наших целевых значений для обоих изображений, при этом окончательная потеря является средним арифметическим суммы этих потерь. Что это? Что ж, если вы посмотрите на исходный код CrossEntropyLoss, вы увидите комментарий: Для каждой мини-партии потери усреднены по наблюдениям (The losses are averaged across observations for each minibatch). Существует также параметр reduction, для которого по умолчанию установлено значение mean (мы до сих пор использовали значение по умолчанию, поэтому раньше вы его не видели). Нужно сохранить это условие, поэтому берем среднее значение совокупных потерь.

Теперь наличие двух загрузчиков данных не составляет особых проблем, но немного усложняет код. Если вы запустите этот код, то можете получить ошибку, потому что пакеты не сбалансированы, так как конечные пакеты выходят из загрузчиков, а это означает, что вам придется написать дополнительный код для решения этой проблемы. Авторы работы о mixup предполагают, что можно заменить загрузчик смешанных данных случайным перемещением входящего пакета. Это можно сделать через torch.randperm():

```
shuffle = torch.randperm(inputs.size(0))
inputs_mix = inputs[shuffle]
targets_mix = targets[shuffle]
```

При использовании смешивания таким образом имейте в виду, что гораздо больше шансов получить *коллизии*, если вы в конечном итоге примените один и тот же параметр к одному и тому же набору изображений, что может привести к снижению точности обучения. Например, вы можете смешать cat1 с fish1 и нарисовать бета-параметр 0,3. Затем, в той же партии, вы извлекаете fish1, который смешивается с cat1 с параметром 0,7! Некоторые

реализации mixup, в частности реализация fast.ai, решают эту проблему, заменяя наши параметры смешивания следующим образом:

```
mix_parameters = torch.max(mix_parameters, 1 - mix_parameters)
```

Этот код гарантирует, что неперемешанная партия всегда будет иметь наивысший компонент при объединении со смесью, то есть устраняет эту потенциальную проблему.

Да, и еще кое-что: мы выполнили преобразование смешивания после преобразования изображения через конвейер. Сейчас наши пакеты — это просто тензоры, которые мы сложили вместе. Нет никакой причины, по которой смешанное обучение должно быть ограничено применительно к изображениям. Его можно использовать для любого типа данных, которые были преобразованы в тензоры, будь то текст, изображение, аудио или что-либо другое.

Мы все еще можем сделать так, чтобы наши маркировки работали лучше. Попробуйте *сглаживание меток* — подход, который является основой современных моделей.

Сглаживание маркировок

Сглаживание маркировок, так же как и смешивание, помогает улучшить производительность модели, делая ее менее уверенной в своих прогнозах. Вместо того чтобы пытаться заставить ее предсказать `1` для спрогнозированного класса (где есть все проблемы, о которых мы говорили в предыдущем разделе), мы изменим ее так, чтобы она прогнозировала 1 минус небольшое значение, *эпсилон*. Мы можем создать новую реализацию функции потерь, которая обернет уже существующую функцию `CrossEntropyLoss`. Как оказалось, написание пользовательской функции потерь — это просто еще один подкласс `nn.Module`:

```
class LabelSmoothingCrossEntropyLoss(nn.Module):
    def __init__(self, epsilon=0.1):
        super(LabelSmoothingCrossEntropyLoss, self).__init__()
        self.epsilon = epsilon

    def forward(self, output, target):
```

```
num_classes = output.size()[-1]
log_preds = F.log_softmax(output, dim=-1)
loss = (-log_preds.sum(dim=-1)).mean()
nll = F.nll_loss(log_preds, target)
final_loss = self.epsilon * loss / num_classes +
             (1-self.epsilon) * nll
return final_loss
```

Когда дело доходит до вычисления функции потерь, то рассчитывается потеря кросс-энтропии согласно реализации `CrossEntropyLoss`. Функция `final_loss` состоит из отрицательного логарифмического правдоподобия, умноженного на 1 минус эпсилон (наша *сглаженная* маркировка), плюс потеря, умноженная на эпсилон, деленная на количество классов. Сглаживается не только маркировка для прогнозируемого класса, равная 1 минус эпсилон, но и другие маркировки, чтобы они приводились не к нулю, а к значению между нулем и эпсилоном.

При обучении такая новая пользовательская функция потерь сможет заменить `CrossEntropyLoss` везде, где мы использовали ее в книге. В сочетании со смешиванием она является невероятно эффективным способом получить немного больше от ваших входных данных.

Теперь мы отвлечемся от аугментации данных, чтобы обратиться к другой горячей теме в современных тенденциях глубокого обучения: генеративно-состязательным сетям.

Компьютер, улучшай!

Десятилетиями мы, программисты, смеялись над телевизионными криминальными шоу, в которых детектив щелкает на кнопке, чтобы размытое изображение с камеры внезапно стало четкой сфокусированной картинкой. Мы высмеивали такие шоу, как «C.S.I.: Место преступления». А сейчас реальность такова, что мы и сами можем делать примерно то же самое. Вот пример этого колдовства на меньшем изображении 256 × 256, масштабированном до 512 × 512, см. рис. 9.4 и 9.5.

Нейронная сеть воссоздает новые детали, чтобы восполнить то, чего на самом деле нет, и эффект может быть впечатляющим. Но как это работает?

Рис. 9.4. Почтовый ящик, разрешение 256 × 256

Рис. 9.5. Почтовый ящик с текстурами, улучшенными с помощью ESRGAN, разрешение 512 × 512

Введение в сверхвысокое разрешение

Вот первая часть очень простой модели со сверхвысоким разрешением. Начнем с того, что она почти такая же, как любая модель, которую вы видели до этого:

```
class OurFirstSRNet(nn.Module):
```

```
  def __init__(self):
      super(OurFirstSRNet, self).__init__()
      self.features = nn.Sequential(
          nn.Conv2d(3, 64, kernel_size=8, stride=4, padding=2),
          nn.ReLU(inplace=True),
          nn.Conv2d(64, 192, kernel_size=2, padding=2),
          nn.ReLU(inplace=True),
          nn.Conv2d(192, 256, kernel_size=2, padding=2),
          nn.ReLU(inplace=True)
      )

  def forward(self, x):
      x = self.features(x)
      return x
```

Если мы передадим случайный тензор через сеть, то получим тензор `shape [1, 256, 62, 62]`; представление изображения было сжато в намного меньший вектор. Давайте теперь представим новый тип слоя, `torch.nn.ConvTranspose2d`. Считайте, что это слой, который инвертирует стандартное преобразование `Conv2d` (со своими собственными обучаемыми параметрами). Добавим новый слой `nn.Sequential`, `upsample` и поместим в простой список этих новых слоев и функций активации `ReLU`. В методе `forward()` передаем входные данные через консолидированный слой после остальных входных данных:

```
class OurFirstSRNet(nn.Module):
  def __init__(self):
      super(OurFirstSRNet, self).__init__()
      self.features = nn.Sequential(
          nn.Conv2d(3, 64, kernel_size=8, stride=4, padding=2),
          nn.ReLU(inplace=True),
          nn.Conv2d(64, 192, kernel_size=2, padding=2),
          nn.ReLU(inplace=True),
          nn.Conv2d(192, 256, kernel_size=2, padding=2),
          nn.ReLU(inplace=True)
      )
      self.upsample = nn.Sequential(
          nn.ConvTranspose2d(256,192,kernel_size=2, padding=2),
          nn.ReLU(inplace=True),
          nn.ConvTranspose2d(192,64,kernel_size=2, padding=2),
          nn.ReLU(inplace=True),
          nn.ConvTranspose2d(64,3, kernel_size=8, stride=4,padding=2),
          nn.ReLU(inplace=True)
      )
  def forward(self, x):
      x = self.features(x)
      x = self.upsample(x)
      return x
```

Если вы сейчас протестируете модель со случайным тензором, то получите тензор точно такого же размера, что и у вошедшего тензора! То, что у нас с вами сейчас получилось, называется автокодировщиком, такой тип сети перестраивает свои входные данные обычно после их сжатия в меньшую размерность. Последующий слой `features` представляет собой кодировщик, который преобразует изображение в тензор размера `[1, 256, 62, 62]`, а слой `upsample` является *декодером*, который преобразует его обратно в исходную форму.

Маркировки для обучения изображения, конечно, будут входными изображениями, но это означает, что мы не можем использовать функции потерь, такие как наша довольно стандартная функция `CrossEntropyLoss`, потому что классов у нас нет! Нужна функция потерь, которая скажет нам, насколько наше выходное изображение отличается от входного изображения, а для этого обычно используются среднеквадратические потери или средние абсолютные потери между пикселами изображения.

Хотя вычисление потерь между пикселами является целесообразным, многие наиболее успешные сети со сверхвысоким разрешением используют аугментированные функции потерь, которые определяют, насколько сгенерированное изображение похоже на исходное, допуская потерю пикселов для лучшей производительности в таких областях, как потеря текстуры и содержимого. Более подробно об этом можно узнать в статьях, перечисленных в разделе «Дополнительные источники».

Вернемся к входным данным того же размера, которые мы ввели, но что, если мы добавим еще одну преобразованную свертку?

```
self.upsample = nn.Sequential(...
nn.ConvTranspose2d(3,3, kernel_size=2, stride=2)
nn.ReLU(inplace=True))
```

Попробуйте! Вы увидите, что выходной тензор вдвое больше, чем входной. Если есть доступ к набору истинных изображений такого размера,

чтобы они могли выступать в качестве маркировок, можно научить сеть принимать изображения размера *x* и создавать изображения размера *2x*. На практике мы выполняем эту повышающую дискретизацию путем масштабирования в два раза больше, чем нужно, а затем добавляем стандартный слой свертки:

```
self.upsample = nn.Sequential(......
nn.ConvTranspose2d(3,3, kernel_size=2, stride=2),
nn.ReLU(inplace=True),
nn.Conv2d(3,3, kernel_size=2, stride=2),
nn.ReLU(inplace=True))
```

Транспонированная свертка склонна добавлять шум и муар по мере увеличения изображения. Увеличивая изображение в два раза, а затем масштабируя его до необходимого размера, мы передаем сети достаточно информации, чтобы сгладить изображения и сделать выходные данные более реалистичными.

Это основы сверхвысокого разрешения. Большинство современных высокопроизводительных сетей сверхвысокого разрешения обучаются по методике генеративно-состязательной сети, захватившей мир глубокого обучения в последние несколько лет.

Введение в GAN

Одной из всеобщих проблем глубокого обучения (или любого приложения машинного обучения) является стоимость производства маркированных данных. В этой книге нам практически удалось избежать этой проблемы, используя выборку наборов данных, которые тщательно маркированы (даже заранее упакованный набор данных для тестов/обучения/верификации). Но в реальности производится большое количество маркированных данных. Методики, о которых вы уже много узнали, например перенос обучения, направлены на то, чтобы делать больше с меньшими затратами. Но иногда вам нужно еще больше, и *генеративно-состязательные сети* (GAN) могут в этом помочь.

Впервые GAN упоминается в 2014 году в статье Яна Гудфеллоу. Эти сети представляют собой новый способ предоставления большего количества данных для обучения нейронных сетей. И подход в основном заключается

в том, что «мы знаем, что вы любите нейронные сети, поэтому мы добавили еще одну».[1]

Фальсификатор и критик

GAN работает по следующему принципу. Две нейросети обучаются вместе. Первая сеть — это *генератор*, который берет случайный шум из векторного пространства входных тензоров и выдает ложные выходные данные. Вторая сеть — это *дискриминатор*, который чередует сгенерированные фальшивые данные с реальными данными. Его задача состоит в том, чтобы смотреть на поступающие входные данные и решать, являются они реальными или фейковыми. Простая концептуальная схема GAN показана на рис. 9.6.

Самое замечательное в GAN — это простота идеи, несмотря на кажущуюся сложность реализации: две сети состязаются между собой, и в процессе обучения изо всех сил стараются друг друга победить. В итоге *генератор* должен выдавать данные, соответствующие *распределению* реальных входных данных, чтобы запутать *дискриминатор*. Как только вы дойдете до этой точки, генератор можно использовать, чтобы генерировать больше данных для любых ваших потребностей. А вот дискриминатор в это время удаляется в бар одиноких нейронных сетей заливать горе.

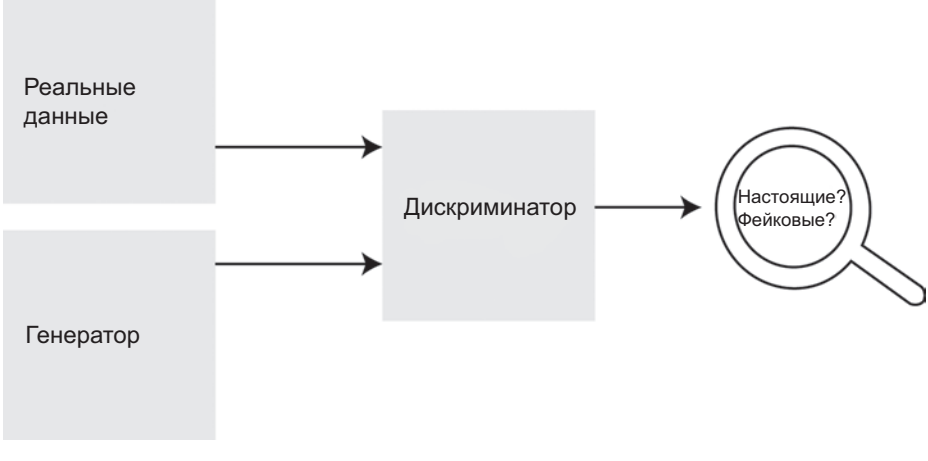

Рис. 9.6. Принцип работы GAN

[1] См. «Generative Adversarial Networks», Ian J. Goodfellow et al. (2014) (https://arxiv.org/abs/1406.2661).

Обучение GAN

Обучение GAN немного сложнее, чем обучение традиционных сетей. Чтобы начать обучение дискриминатора, во время цикла обучения сначала нужно использовать реальные данные. Мы рассчитываем потери дискриминатора (используя BCE, поскольку у нас есть только два класса: реальный или фейковый), а затем выполняем обратный проход, чтобы обновить параметры дискриминатора. Но на этот раз мы *не* вызываем оптимизатор для обновления. Вместо этого мы генерируем пакет данных из нашего генератора и передаем его через модель. Мы рассчитываем потери и делаем *еще один* обратный проход, так чтобы в этот момент цикл обучения рассчитал потери двух проходов через модель. Теперь мы вызываем оптимизатор для обновления на основе этих аккумулированных градиентов.

Во второй части тренировки переходим к генератору. Мы предоставляем генератору доступ к дискриминатору, а затем генерируем новый пакет данных (которые, как утверждает генератор, реальны) и тестируем на дискриминаторе. Формируем функцию потери от этих выходных данных, где каждая точка данных, которую дискриминатор считает фальшивой, считается *неправильным* ответом — потому что мы пытаемся ее обмануть, — а затем выполняем стандартный обратный проход/оптимизацию.

Вот общая реализация в PyTorch. Обратите внимание, что генератор и дискриминатор являются обычными стандартными нейронными сетями, поэтому теоретически могут генерировать изображения, текст, аудио или любой другой тип данных и принадлежать к любому из типов сетей, которые вы уже видели:

```
generator = Generator()
discriminator = Discriminator()

# Set up separate optimizers for each network
generator_optimizer = ...
discriminator_optimizer = ...

def gan_train():
  for epoch in num_epochs:
    for batch in real_train_loader:
      discriminator.train()
      generator.eval()
      discriminator.zero_grad()
```

```
preds = discriminator(batch)
real_loss = criterion(preds, torch.ones_like(preds))
discriminator.backward()

fake_batch = generator(torch.rand(batch.shape))
fake_preds = discriminator(fake_batch)
fake_loss = criterion(fake_preds, torch.zeros_like(fake_preds))
discriminator.backward()

discriminator_optimizer.step()
discriminator.eval()
generator.train()
generator.zero_grad()

forged_batch = generator(torch.rand(batch.shape))
forged_preds = discriminator(forged_batch)
forged_loss = criterion(forged_preds, torch.ones_like(forged_preds))

generator.backward()
generator_optimizer.step()
```

Обратите внимание, что здесь очень помогает гибкость PyTorch. При отсутствии специального цикла обучения, который в основном предназначен для более стандартного обучения, мы привыкли создавать новый цикл и знаем все этапы, которые нужно включить. В некоторых других фреймворках обучение GAN представляется более хлопотным. И это важно, потому что обучение GAN — само по себе достаточно сложная задача и без стоящих у него на пути фреймворков.

Схлопывание мод распределения

В идеальном мире во время обучения происходит следующее: дискриминатор сначала будет хорошо обнаруживать фальшивые данные, поскольку обучается на реальных данных, тогда как генератору дается доступ только к дискриминатору, а не к самим реальным данным. В конце концов генератор научится обманывать дискриминатор, а затем улучшит свою работу, чтобы соответствовать распределению данных и многократно создавать фальшивые данные, которые проскальзывают мимо критика.

Но кое-что мешает многим архитектурам GAN; это *схлопывание мод распределения* (mode collapse). Если у наших реальных данных есть три типа, то, возможно, генератор начнет генерировать первый тип и, возможно, научится делать это довольно хорошо. Дискриминатор может решить, что

все, что похоже на первый тип, на самом деле является фейком, даже сам реальный пример, и тогда генератор начинает генерировать нечто похожее на третий тип. Дискриминатор начинает отклонять все выборки третьего типа, и генератор выбирает из реальных примеров для генерации. Цикл продолжается бесконечно; генератору никогда не удается генерировать выборки из всего распределения.

Снижение схлопывания мод распределения является ключевой проблемой производительности при использовании GAN и предметом постоянных исследований. Некоторые подходы включают добавление к сгенерированным данным показателя сходства, так чтобы потенциальный коллапс можно было обнаружить и предотвратить. Для этого сохраняют буфер воспроизведения сгенерированных изображений, чтобы дискриминатор не переобучался на самом последнем пакете сгенерированных изображений, разрешая добавление фактических маркировок из реального набора данных в сеть генератора и т. д. Завершаем мы этот раздел приложением GAN, которое выполняет сверхчеткое разрешение.

ESRGAN

ESRGAN (Enhanced Super-Resolution Generative Adversarial Network) — это сеть, разработанная в 2018 году, которая дает впечатляющие результаты сверхвысокого разрешения. Генератор представляет собой серию блоков сверточных сетей с комбинацией остаточного и плотного слоев (смесь как ResNet, так и DenseNet), соединенных со слоями `Batch Norm`, которые были удалены, так как они создают артефакты в изображениях с повышенной дискретизацией. Что касается дискриминатора, то вместо того, чтобы просто выдавать результат о том, *настоящее это изображение или фальшивое*, он прогнозирует вероятность того, что реальное изображение будет относительно более реалистичным, чем фальшивое, и это помогает заставить модель производить более лаконичные результаты.

Запуск ESRGAN

Чтобы посмотреть на ESRGAN в действии, загрузите код из репозитория GitHub (https://github.com/xinntao/ESRGAN).

Используйте `git`:

```
git clone https://github.com/xinntao/ESRGAN
```

Далее нужно загрузить веса, чтобы использовать модель без обучения. Используя ссылку на Google Диск в файле README, загрузите файл RRDB_ESRGAN_x4.pth и поместите его в ./models. Попробуем увеличить разрешение уменьшенного изображения нашей Гельветики в коробке, но вы можете поместить в каталог ./LR и любое другое изображение. Запустите готовый скрипт *test.py*, и вы увидите, что изображения генерируются и сохраняются в каталоге *results*.

На этом со сверхвысоким разрешением покончено, но с изображениями мы еще не закончили.

Новые приключения в распознавании образов

Классификации изображений в главах 2–4 имели одну общую черту: мы определили, что изображение принадлежит к одному классу — кошке или рыбке. И очевидно, что в реальных приложениях классов будет гораздо больше. Но мы также предполагаем, что на картинках могут быть изображены и кошка с рыбкой (что может быть плохой новостью для рыбки) или любой из классов, которые мы ищем. На изображениях могут быть представлены два человека на сцене, машина и лодка, и нужно не только определить, есть ли эти объекты на изображении, но и *где* они находятся. Есть два основных способа: *обнаружение объектов* и *сегментация*. Мы рассмотрим оба, а затем на конкретных примерах узнаем, как с помощью PyTorch компания Facebook реализовала Faster R-CNN и Mask R-CNN.

Обнаружение объектов

Давайте посмотрим на кошку в коробке. Требуется, чтобы сеть поместила кошку в коробке в другую коробку! В частности, мы хотим получить *ограничивающий прямоугольник*, который охватывает все изображение, которое модель считает кошкой (рис. 9.7).

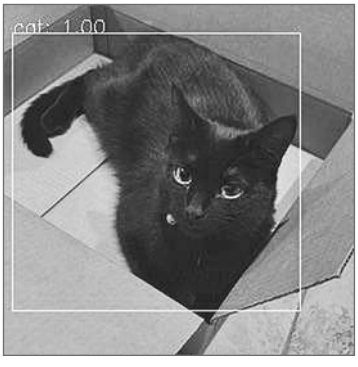

Рис. 9.7. Кошка в коробке в ограничивающем прямоугольнике

Но как заставить наши сети решить эту задачу? Помните, что нейросети могут прогнозировать все, что вы от них хотите. Что, если наряду с нашим прогнозом класса создать еще четыре выхода? В модели CATFISH у нас был бы слой `Linear` с выходным размером `6`, а не `2`. Дополнительные четыре выхода будут определять прямоугольник, используя координаты x_1, x_2, y_1, y_2.

Вместо того чтобы просто предоставлять изображения в качестве обучающих данных, нам придется аугментировать их с помощью ограничивающих прямоугольников, оставив при этом модели возможность чему-то обучаться. Функция потерь теперь будет представлять собой комбинированную потерю кросс-энтропийной потери прогноза по классу и среднеквадратическую потерю для ограничивающих прямоугольников.

Никакой магии! Мы просто проектируем модель, которая даст то, что нам нужно, вводим данные, которые содержат достаточно информации для составления прогнозов и обучения, и включаем функцию потерь, которая сообщает нашей сети, насколько хорошо или плохо она справляется.

Альтернативой ограничивающих прямоугольников является *сегментация*. Вместо них наша сеть выводит маску изображения с тем же размером ввода; пикселы в маске окрашены в зависимости от того, в какой класс они попадают. Например, трава может быть зеленой, дороги — фиолетовыми, машины — красными и т. д.

Вы абсолютно правы, если считаете, что при выводе изображения мы будем использовать архитектуру такого же типа, что и в разделе про

сверхвысокое разрешение. Эти два способа имеют между собой много общего, и один из типов моделей, который стал популярным за последние несколько лет, — это архитектура U-Net, показанная на рис. 9.8.[1]

Как видите, классическая архитектура U-Net представляет собой набор сверточных блоков, которые уменьшают изображение, и еще один набор сверток, которые снова масштабируют его до целевого изображения. Однако ключом U-Net являются линии, проходящие от левых блоков к их аналогам справа, которые соединяются с выходными тензорами при обратном масштабировании до нужного размера. Эти соединения позволяют передавать информацию из сверточных блоков более высокого уровня, сохраняя детали, которые могут быть удалены, поскольку сверточные блоки уменьшают входное изображение. Вы встретите архитектуры на базе U-Net на всех соревнованиях по сегментации Kaggle, а это в некотором смысле доказывает, что эта структура хороша для сегментации. Другой метод — наш старый друг, перенос обучения. В этом подходе первая часть U взята из предварительно обученной модели, например ResNet или Inception, а другая сторона U и пропущенные соединения добавляются поверх обученной сети и настраиваются в обычном режиме.

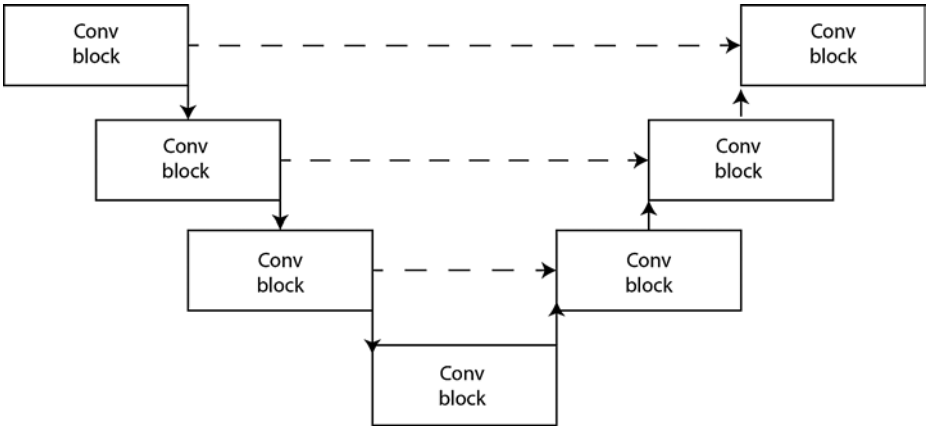

Рис. 9.8. Упрощенная архитектура U-Net

[1] См. «U-Net: Convolutional Networks for Biomedical Image Segmentation», Olaf Ronneberger et al. (2015) (https://arxiv.org/abs/1505.04597).

Давайте рассмотрим некоторые существующие предварительно обученные модели, которые обеспечивают самое современное обнаружение объектов и сегментацию прямо из Facebook.

Faster R-CNN и Mask R-CNN

Facebook Research выпустила библиотеку maskrcnn-benchmark, которая содержит эталонные реализации как алгоритмов обнаружения объектов, так и алгоритмов сегментации.

Давайте установим библиотеку и добавим код для генерации прогнозов. На момент написания этой книги самым простым способом построения моделей было использование Docker (с выходом PyTorch 1.2 ситуация может измениться). Скопируйте репозиторий по ссылке https://github.com/facebook research/maskrcnn-benchmark и добавьте скрипт predict.py в каталог demo, чтобы настроить конвейер прогнозирования с использованием бэкбона ResNet-101:

```
import matplotlib.pyplot as plt

from PIL import Image
import numpy as np
import sys
from maskrcnn_benchmark.config import cfg
from predictor import COCODemo

config_file = "../configs/caffe2/e2e_faster_rcnn_R_101_FPN_1x_caffe2.yaml"

cfg.merge_from_file(config_file)
cfg.merge_from_list(["MODEL.DEVICE", "cpu"])

coco_demo = COCODemo(
    cfg,
    min_image_size=500,
    confidence_threshold=0.7,
)

pil_image = Image.open(sys.argv[1])
image = np.array(pil_image)[:, :, [2, 1, 0]]
predictions = coco_demo.run_on_opencv_image(image)
predictions = predictions[:,:,::-1]

plt.imsave(sys.argv[2], predictions)
```

В этом коротком скрипте сначала нужно настроить предиктор `COCODemo`. Важно проследить, чтобы передалась конфигурация, которая устанавливает Faster R-CNN, а не Mask R-CNN (она производит сегментированные выходные данные). Затем нужно открыть файл изображения, заданный в командной строке, но перед этим преобразуем его в формат `BGR` вместо формата `RGB`, поскольку предиктор обучается на изображениях OpenCV, а не на изображениях PIL, которые мы использовали до этого. Наконец, мы используем `imsave`, чтобы записать массив `predictions` (исходное изображение плюс ограничивающий прямоугольник) в новый файл, также указанный в командной строке. Скопируйте файл тестового изображения в каталог `demo`, чтобы создать Docker-образ:

```
docker build docker/
```

Мы запускаем скрипт из Docker-контейнера и создаем вывод, как на рис. 9.7 (фактически я использовал библиотеку для того, чтобы сгенерировать это изображение). Попробуйте поэкспериментировать с разными значениями `confidence_threshold` и разными картинками. Также можете переключиться на конфигурацию `e2e_mask_rcnn_R_101_FPN_1x_caffe2.yaml`, чтобы испытать Mask R-CNN и сгенерировать маски сегментации.

Чтобы обучать собственные данные на моделях, нужно будет предоставить свой набор данных, который обеспечивает маркировки ограничивающего прямоугольника для каждого изображения. Библиотека предоставляет вспомогательную функцию `BoxList`. Вот каркасная реализация набора данных, которые можно использовать для начала:

```
from maskrcnn_benchmark.structures.bounding_box import BoxList

class MyDataset(object):
    def __init__(self, path, transforms=None):
        self.images = # настраивает список изображений
        self.boxes = # читает в блоках
        self.labels = # читает в метках

    def __getitem__(self, idx):
        image = # Получить изображение PIL из self.images
        boxes = # Создать список массивов, по одному на каждый блок
        в формате x1, y1, x2, y2
        labels = # Метки, которые соответствуют блокам

        boxlist = BoxList(boxes, image.size, mode="xyxy")
        boxlist.add_field("labels", labels)
```

```
        if self.transforms:
            image, boxlist = self.transforms(image, boxlist)

        return image, boxlist, idx

    def get_img_info(self, idx):
        return {"height": img_height, "width": img_width
```

Затем вам нужно будет добавить новый созданный набор данных в maskrcnn_benchmark/data/ datasets/init.py и maskrcnn_benchmark/config/paths_catalog.py. Обучение можно проводить с использованием прилагаемого скрипта train_net.py в репозитории. Имейте в виду, что может потребоваться уменьшить размер пакета для обучения любой из этих сетей на одном GPU.

На этом мы завершаем раздел об обнаружении объектов и сегментации. Узнать больше можно в разделе «Дополнительные источники», в том числе информацию о чудесной архитектуре «You Only Look Once (YOLO)». А пока посмотрим, как умышленно вывести модель из строя.

Состязательные семплы

Возможно, вы уже читали статьи об изображениях, которые могут как-то помешать нормальному распознаванию изображений. Если человек подносит изображение к камере, нейронная сеть думает, что видит панду или что-то вроде того. Такие изображения с подвохом известны как состязательные семплы и представляют собой интересные способы обнаружения ограничений ваших архитектур.

Создать состязательные семплы не так уж сложно, особенно если есть доступ к модели. Вот простая нейронная сеть, которая классифицирует изображения из распространенного набора данных CIFAR-10. В этой модели нет ничего особенного, поэтому можете смело заменить ее на AlexNet, ResNet или любую другую сеть, описанную в книге:

```
class ModelToBreak(nn.Module):
    def __init__(self):
        super(ModelToBreak, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
```

```
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Обучив сеть на CIFAR-10, мы можем получить прогноз для изображения на рис. 9.9. Надеюсь, обучение прошло достаточно успешно и сеть смогла распознать лягушку (если нет, то требуется чуть дольше времени). А теперь изменим изображение лягушки настолько, чтобы нейронная сеть запуталась и подумала, что это что-то другое, хотя мы все еще знаем, что это лягушка.

Для этого используем метод атаки *Fast gradient sign method*.[1] Идея состоит в том, чтобы взять изображение, которое мы хотим неправильно классифицировать, и запустить его через модель, как обычно, что дает нам выходной тензор. Обычно для прогнозов мы смотрим, какое из значений тензора было самым высоким, и применяем его в качестве индекса для наших классов, используя `argmax()`. Но на этот раз мы притворимся, что снова обучаем сеть и выполняем обратное распространение результатов через модель в обратном направлении, получая изменения градиента модели относительно исходных входных данных (в случае нашего изображения лягушки). Таким образом мы создаем новый тензор, который просматривает эти градиенты и заменяет запись +1, если градиент положительный, и –1, если градиент отрицательный.

Это дает нам направление движения, в котором это изображение проверяет границы доступного решения модели. Затем умножаем на небольшой скаляр (*эпсилон*), чтобы создать маску, которую мы затем добавляем к исходному изображению, создавая состязательный пример.

[1] См. «Explaining and Harnessing Adversarial Examples», Ian Goodfellow et al. (2014) (https://arxiv.org/abs/1412.6572).

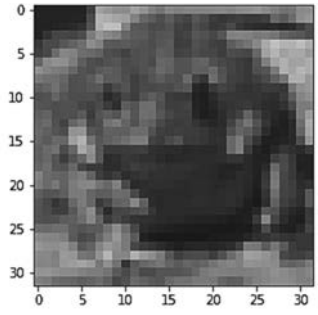

Рис. 9.9. Пример лягушки

Вот простой метод PyTorch, который возвращает тензоры метода быстрой смены знака градиента (FGSM) входного пакета при наличии маркировки пакета, плюс модель и функция потерь, использованная для оценки модели:

```
def fgsm(input_tensor, labels, epsilon=0.02, loss_function, model):
    outputs = model(input_tensor)
    loss = loss_function(outputs, labels)
    loss.backward(retain_graph=True)
    fsgm = torch.sign(inputs.grad) * epsilon
    return fgsm
```

Эпсилон обычно можно найти экспериментальным путем. Экспериментируя с изображениями, я обнаружил, что значение `0.02` хорошо подходит для этой модели, но вы также можете использовать что-то вроде сетки или случайного поиска, чтобы найти значение, которое превращает лягушку в корабль!

Запустив эту функцию на нашей лягушке и модели, мы получим маску, которую можно добавить к нашему исходному изображению, чтобы сгенерировать состязательные семплы. Посмотрите, что получилось на рис. 9.10!

```
model_to_break = # load our model to break here
adversarial_mask = fgsm(frog_image.unsqueeze(-1),
                        batch_labels,
                        loss_function,
                        model_to_break)
adversarial_image = adversarial_mask.squeeze(0) + frog_image
```

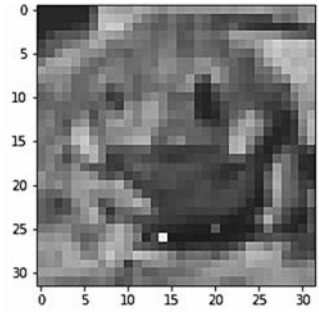

Рис. 9.10. Наша состязательная лягушка

Очевидно, что наш глаз все еще воспринимает созданный нами образ как лягушку. (Если для вас это не так, то, возможно, вы — нейронная сеть. Срочно пройдите эмпатический тест Войта-Кампфа.) Но что произойдет, если мы получим прогноз от модели на этом новом изображении?

```
model_to_break(adversarial_image.unsqueeze(-1))
# искать в метках через argmax()
>> 'cat'
```

Мы победили модель. Но такая ли это большая проблема, как кажется на первый взгляд?

Black-Box-атаки

Возможно, вы заметили, что для создания изображения, которое обманывает классификатор, нам нужно много знать об используемой модели. У нас есть вся структура модели, а также функция потерь, которая использовалась при обучении модели, и чтобы получить градиенты, нам нужно делать прямые и обратные проходы. Это классический пример того, что в компьютерной безопасности известно как *white-box*-атака, где мы можем заглянуть в любую часть кода, выяснить, что там происходит, и использовать это в своих целях.

Так имеет ли это значение? В конце концов, большинство моделей, с которыми вы столкнетесь в интернете, не позволят вам заглянуть внутрь. Действительно ли возможна *black-box*-атака, когда все, что у вас есть, это входные и выходные данные? К сожалению, да. Допустим, у нас есть набор входных и набор выходных данных для противопоставления. Выходные

данные — это *маркировки*, и можно использовать целевые запросы моделей, чтобы обучить новую модель, которую вы можете использовать в качестве локального прокси-сервера, и проводить white-box-атаки. Как и в случае с переносом обучения, атаки на прокси-модель могут эффективно работать на реальной модели. Мы обречены?

Защита от состязательных атак

Как мы можем защититься от этих атак? Например, для чего-то вроде определения того, кто на изображении — кошка или рыбка, это, вероятно, не конец света, но для беспилотных систем, приложений для обнаружения рака и т. д. это может буквально означать разницу между жизнью и смертью. Успешная защита от всех типов подобных атак все еще остается областью исследований, но основные моменты до сих пор включают дистилляцию и валидацию.

Кажется, что метод дистилляции модели с помощью использования ее для обучения другой модели работает. Использование сглаживания маркировок с новой моделью, как описано выше, также работает. Если сделать модель менее уверенной в своих решениях, это поможет сгладить градиенты, делая атаку на базе градиента менее эффективной.

Более действенный подход — вернуться к некоторым моментам начального этапа компьютерного зрения. Если мы проведем контроль ввода на входных данных, мы, возможно, сможем предотвратить попадание в модель изображения состязательного образа. В предыдущем примере сгенерированное изображение атаки имеет несколько пикселов, которые совершенно не соответствуют тому, на что мы рассчитываем, когда видим лягушку. В зависимости от домена у нас может быть фильтр, который допускает только изображения, которые проходят некоторую фильтрацию.

Теоретически вы могли бы заставить нейронную сеть сделать так же, потому что тогда хакеры должны будут попытаться взломать две разные модели одним и тем же изображением!

Теперь с изображениями действительно покончено. Но давайте посмотрим на некоторые события, которые произошли за последние пару лет в текстовых сетях.

Больше, чем кажется: архитектура Transformer

Перенос обучения стал важной особенностью, позволившей сетям, основанным на анализе изображений, стать такими эффективными и распространенными в течение последнего десятилетия, но текст всегда был более крепким орешком. Однако за последние несколько лет были предприняты некоторые важные шаги, которые раскрывают возможности использования переноса обучения в тексте для выполнения всевозможных задач, таких как генерация, классификация и ответы на вопросы.

Мы также видели, что новый тип архитектуры начинает занимать центральное место: мы говорим о *сети Transformer*. Эти сети появились не от Cybertron, но эти технологии лежат в основе самых мощных текстовых сетей, которые мы когда-либо видели: речь идет о модели GPT-2 от OpenAI, выпущенной в 2019 году, которая демонстрирует впечатляющее качество генерируемого текста — настолько, что OpenAI изначально скрывала большую версию модели, чтобы предотвратить ее использование в противозаконных целях. Мы рассмотрим общую теорию Transformer, а затем углубимся в то, как использовать реализации GPT-2 и BERT от Hugging Face.

Механизмы внимания

Первым шагом на пути к архитектуре Transformer был механизм внимания, который первоначально был представлен в RNN для помощи в преобразованиях последовательности в последовательность, таких как трансляция.[1]

Задачи, которое мы пытались решить с помощью внимания, заключались в трудности перевода предложений, таких как «Кошка села на коврик, и она мурлыкала» (The cat sat on the mat and she purred). Мы-то с вами знаем, что *она* (*she*) в этом предложении относится к кошке, но RNN это сложно понять. Здесь может быть скрытое состояние, о котором мы говорили в главе 5, но к тому времени, когда мы доберемся до слова *она*

[1] См. «Neural Machine Translation by Jointly Learning to Align and Translate», Dzmitry Bahdanau et al. (2014) (https://arxiv.org/abs/1409.0473).

(*she*), у нас уже будет много временных шагов и скрытое состояние для каждого шага!

Итак, что делает *внимание*. Оно добавляет дополнительный набор обучаемых весов, прикрепленных к каждому временному шагу, который фокусирует сеть на определенной части предложения. Веса обычно передаются через слой `softmax`, чтобы сгенерировать вероятности для каждого шага, а затем скалярное произведение весов внимания рассчитывается с учетом предыдущего скрытого состояния. На рис. 9.11 представлена упрощенная версия этого процесса для нашего предложения.

Веса гарантируют, что когда скрытое состояние объединится с текущим состоянием, *кошка* (*cat*) станет основной частью определения выходного вектора во временном шаге для слова *она* (*she*), что обеспечит полезный контекст для перевода, например, на французский.

Мы не будем вдаваться в детали о том, как *внимание* может работать в конкретной реализации, но мы знаем, что концепция имела настолько широкие возможности, что положила начало впечатляющему росту и точности Google Translate еще в середине 2010-х годов. Но многое еще было впереди.

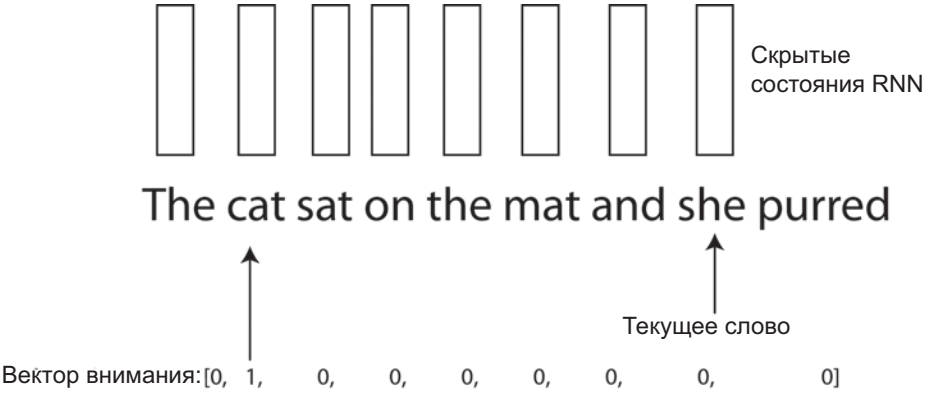

Рис. 9.11. Вектор внимания, указывающий на кошку

Все, что нужно, — это внимание

В передовой статье «Attention Is All You Need»[1] исследователи Google указали, что мы потратили все это время на привлечение внимания к и без того медленной сети на базе RNN (во всяком случае, по сравнению с CNN или линейными элементами). Что, если бы RNN была нам не нужна? В документе показано, что с помощью стековых кодировщиков и декодеров на базе внимания вы могли бы создать модель, которая вообще не зависела бы от скрытого состояния RNN, что привело бы к появлению более крупного и быстрого Transformer, который сегодня господствует в глубоком обучении на базе распознавания текста.

Ключевая идея заключалась в том, чтобы использовать то, что авторы назвали механизмом *multihead attention*: он параллелизует шаг *внимания* по всем входным данным, используя группу слоев `Linear`. Благодаря этому, а также заимствуя некоторые приемы подключения у ResNet, Transformer быстро начал заменять RNN для многих текстовых приложений. Два важных релиза Transformer, BERT и GPT-2, представляют текущее состояние этой технологии на момент публикации этой книги.

К счастью для нас, существует библиотека от Hugging Face, которая реализует оба решения в PyTorch. Ее можно установить с помощью `pip` или `conda`, и вам также понадобится `git clone` самого репозитория, так как позже мы будем использовать некоторые скрипты утилит.

```
pip install pytorch-transformers
conda install pytorch-transformers
```

Сначала разберемся с BERT.

BERT

Модель BERT (*Bidirectional Encoder Representations from Transformers*) от Google, выпущенная в 2018 году, была одним из первых успешных примеров применения переноса обучения мощной модели в тестировании. BERT сама по себе представляет массивную модель на основе Transformer (весом в 110 миллионов параметров в самой маленькой версии), предварительно обученную на Википедии и наборе данных Book-Corpus. Проблема,

[1] https://arxiv.org/abs/1706.03762

с которой традиционно сталкиваются и Transformer, и сверточные сети при работе с текстом, заключается в том, что поскольку они видят все данные одновременно, им сложно выучить временную структуру языка. BERT справляется с этим на этапе предварительного обучения, выполняя маскирование 15 % ввода текста случайным образом и заставляя модель прогнозировать маскированные части. Несмотря на концептуальную простоту, сочетание огромного размера 340 миллионов параметров в самой большой модели с архитектурой Transformer привело к новому уровню результатов для целого ряда оценок, связанных с текстом.

Конечно, несмотря на то что BERT был создан Google с помощью TensorFlow, существуют его реализации и для PyTorch. Давайте кратко рассмотрим одну из них.

FastBERT

Самый простой способ начать использовать модель BERT в ваших собственных приложениях для классификации — это использовать библиотеку *FastBERT*, которая смешивает репозиторий Hugging Face с API fast.ai (о чем вы узнаете чуть подробнее, когда мы перейдем к ULMFiT). Ее можно установить через `pip` обычным способом:

```
pip install fast-bert
```

Вот скрипт, который можно использовать для точной настройки BERT в нашем наборе данных Sentiment140 Twitter. Мы использовали его в главе 5:

```
import torch
import logger

from pytorch_transformers.tokenization import BertTokenizer
from fast_bert.data import BertDataBunch
from fast_bert.learner import BertLearner
from fast_bert.metrics import accuracy

device = torch.device('cuda')
logger = logging.getLogger()
metrics = [{'name': 'accuracy', 'function': accuracy}]

tokenizer = BertTokenizer.from_pretrained
                ('bert-base-uncased',
                  do_lower_case=True)
```

```
databunch = BertDataBunch([PATH_TO_DATA],
                          [PATH_TO_LABELS],
                          tokenizer,
                          train_file=[TRAIN_CSV],
                          val_file=[VAL_CSV],
                          test_data=[TEST_CSV],
                          text_col=[TEST_FEATURE_COL], label_col=[0],
                          bs=64,
                          maxlen=140,
                          multi_gpu=False,
                          multi_label=False)


learner = BertLearner.from_pretrained_model(databunch,
                        'bert-base-uncased',
                        metrics,
                        device,
                        logger,
                        is_fp16=False,
                        multi_gpu=False,
                        multi_label=False)

learner.fit(3, lr='1e-2')
```

После импорта мы настраиваем объекты `device`, `logger` и `metrics`, которые требуются для объекта `BertLearner`. Затем мы создаем `BERTTokenizer` для токенизации входных данных, и в этой базе мы будем использовать модель `bert-base-uncased` (которая имеет 12 слоев и 110 миллионов параметров). Далее нам нужен объект `BertDataBunch`, который содержит пути к обучению, валидации и контрольным наборам данных, где можно найти маркировки, размер нашего пакета и максимальную длину входных данных, что в нашем случае достаточно просто, потому что это может быть только длина твита, то есть 140 символов. Теперь необходимо настроить модель BERT с помощью метода `BertLearner.from_pretrained_model`. Он передает наши входные данные, тип нашей модели BERT, объекты `metric`, `device` и `logger`, которые мы задали в начале скрипта, и, наконец, некоторые флаги, чтобы отключить опции обучения, которые нам не нужны, но они не даны по умолчанию для сигнатуры метода.

Наконец, метод `fit()` отвечает за настройку модели BERT, используя собственный внутренний цикл обучения. В этом примере мы обучаем модель три эпохи со скоростью `1e-2`. Доступ к обученной модели PyTorch можно получить позже через `learner.model`. Вот как начать работать с BERT. А теперь — на соревнование.

GPT-2

Пока Google спокойно работал над BERT, OpenAI работал над собственной версией текстовой модели на базе Transformer. Вместо использования маскирования, чтобы заставить модель изучать структуру языка, модель ограничивает механизм внимания в архитектуре, чтобы просто прогнозировать следующее слово в последовательности, аналогично RNN, описанной в главе 5. В результате GPT осталась, и ее опередила впечатляющая производительность BERT, но в 2019 году OpenAI нанесла ответный удар, выпустив GPT-2, новую версию модели, которая обновила планку для генерации текста.

Магия GPT-2 заключается в масштабе: модель обучается на тексте с более чем 8 миллионами веб-сайтов, а самый большой вариант GPT-2 имеет 1,5 миллиарда параметров. И хотя она по-прежнему уступает BERT по таким параметрам, как, например, вопросы/ответы или другие задачи NLP, ее способность создавать невероятно реалистичный текст из базового ввода привела к тому, что OpenAI заблокировала полноразмерную модель из опасений, что она будет использоваться во вред. Тем не менее они выпустили меньшие версии модели со 117 и 340 миллионами параметров.

Вот пример выходных данных, которые может генерировать GPT-2. Все, что написано курсивом, было написано моделью 340M GPT-2:

> Джек и Джилл отправились в гору *на велосипедную прогулку. Небо было серо-белым, а ветер дул, вызывая сильный снегопад. Ехать по склону было действительно трудно, мне пришлось наклониться вперед и приложить усилия. Но затем настал момент свободы, который я никогда не забуду: я остановил велосипед на склоне горы и был посреди всего этого. У меня не было времени сказать ни слова, но я наклонился вперед и коснулся тормозов, и велосипед поехал.*

Помимо перехода от Джека и Джилл к «я», это впечатляющий фрагмент генерации текста. Короткие фрагменты текста иногда невозможно отличить от текста, созданного человеком. Читая более длинные фрагменты генерируемого текста, действительно можно узнать почерк машины, но это очень впечатляюще, и возможно, такая машина прямо сейчас пишет твиты и комментарии на Reddit. Давайте посмотрим, как это сделать с помощью PyTorch.

Генерация текста с помощью GPT-2

Как и BERT, официальный релиз GPT-2 от OpenAI является моделью TensorFlow. Hugging Face тоже выпустил версию PyTorch, которая содержится в той же библиотеке (`pytorch-transformers`). Тем не менее зарождающаяся экосистема была построена вокруг исходной модели TensorFlow, которой в настоящее время просто не существует в версии PyTorch. Итак, мы смошенничаем один-единственный раз: будем использовать некоторые библиотеки TensorFlow для настройки модели GPT-2, а затем экспортируем веса и импортируем их в версию модели PyTorch.

Чтобы избежать лишних настроек, мы также выполняем все операции TensorFlow в блокноте Colab! Давайте приступим.

Откройте новый блокнот Google Colab и установите библиотеку *gpt-2-simple* от Max Woolf, которая обертывает настройку GPT-2 в одном пакете. Установите ее, добавив в ячейку следующее:

```
!pip3 install gpt-2-simple
```

Далее вам нужен текст. В этом примере я использую общедоступный текст из сборника рассказов П. Г. Вудхауса. Я также не буду заниматься дальнейшей обработкой текста после его загрузки с веб-сайта Project Gutenberg с помощью `wget`:

```
!wget http://www.gutenberg.org/cache/epub/8164/pg8164.txt
```

Теперь можно использовать библиотеку для обучения. Сначала убедитесь, что ноутбук подключен к графическому процессору (Runtime → Change Runtime Type), а затем запустите код в ячейке:

```
import gpt_2_simple as gpt2

gpt2.download_gpt2(model_name="117M")

sess = gpt2.start_tf_sess()
gpt2.finetune(sess,
              "pg8164.txt",model_name="117M",
              steps=1000)
```

Замените текстовый файл используемым вами файлом. Когда модель обучается, она будет выдавать семпл каждые сто шагов. В моем случае было интересно увидеть, как она превращается из модели, выдающей

что-то похожее на сценарии пьес Шекспира, в нечто, что в конечном итоге приближается к прозе Вудхауса. Скорее всего, обучение в течение 1000 эпох длится час или два, так что займитесь чем-то более интересным, пока жужжат облачные графические процессоры.

По окончании процесса выведите веса из Colab в свою учетную запись Google Диска, чтобы их можно было загружать в любое место, где вы запускаете PyTorch:

```
gpt2.copy_checkpoint_to_gdrive()
```

Далее вам будет предложено открыть новую веб-страницу и скопировать код аутентификации в блокнот. Сделайте это, и веса будут собраны и сохранены на вашем Google Диске как *run1.tar.gz*.

Теперь на экземпляре или в блокноте, где вы запустили PyTorch, загрузите этот tar-файл и распакуйте его. Нам нужно переименовать пару файлов, чтобы сделать эти веса совместимыми с повторной реализацией GPT-2 Hugging Face:

```
mv encoder.json vocab.json
mv vocab.bpe merges.txt
```

Теперь нужно конвертировать сохраненные веса TensorFlow в совместимые с PyTorch. Для этого есть удобный репозиторий `pytorch-transformers`:

```
python [REPO_DIR]/pytorch_transformers/convert_gpt2_checkpoint_to_pytorch.py
--gpt2_checkpoint_path [SAVED_TENSORFLOW_MODEL_DIR]
--pytorch_dump_folder_path [SAVED_TENSORFLOW_MODEL_DIR]
```

Создание нового экземпляра модели GPT-2 можно выполнить в коде:

```
from pytorch_transformers import GPT2LMHeadModel

model = GPT2LMHeadModel.from_pretrained([SAVED_TENSORFLOW_MODEL_DIR])
```

Или, просто чтобы поиграть с моделью, используйте скрипт *run_gpt2.py*, чтобы получить подсказку, куда вводить текст и где получать сгенерированные семплы из модели PyTorch:

```
python [REPO_DIR]/pytorch-transformers/examples/run_gpt2.py
--model_name_or_path [SAVED_TENSORFLOW_MODEL_DIR]
```

Вероятно, обучение GPT-2 в ближайшее время станет легче, так как Hugging Face включает понятный API для всех моделей в репозитории, но проще всего прямо сейчас начать использовать метод TensorFlow.

Сейчас BERT и GPT-2 являются самыми популярными нейросетями в области текстового обучения, но прежде чем мы подведем итоги, давайте рассмотрим темную лошадку современных моделей: ULMFiT.

ULMFiT

В отличие от таких гигантов, как BERT и GPT-2, ULMFiT основана на старой доброй RNN. Здесь только AWD-LSTM, архитектурой, изначально созданной Стивеном Мерити. Эта сеть, обученная на наборе данных WikiText-103, оказалась пригодной для переноса, и, несмотря на *старый* тип архитектуры, доказала, что способна конкурировать с BERT и GPT-2 в области классификации.

Хотя ULMFiT, по сути, является еще одной моделью, которую можно загрузить и использовать в PyTorch так же, как и любую другую модель, ее домом является библиотека fast.ai, находящаяся на вершине PyTorch и предоставляющая множество полезных абстракций, которые можно понять и овладеть ими для быстрой продуктивной работы с глубоким обучением. Для этого мы рассмотрим, как использовать ULMFiT с библиотекой fast.ai на примере набора данных Twitter, который мы использовали в главе 5.

Сначала мы используем Data Block API библиотеки fast.ai для подготовки данных к настройке LSTM:

```
data_lm = (TextList
           .from_csv("./twitter-data/",
           'train-processed.csv', cols=5,
           vocab=data_lm.vocab)
           .split_by_rand_pct()
           .label_from_df(cols=0)
           .databunch())
```

Этот код очень похож на вспомогательные методы `torchtext` из главы 5 и просто делает то, что fast.ai называет `databunch`, откуда его модели и процедуры обучения могут легко получать данные. Далее мы создаем модель, но в fast.ai это делается немного по-другому. Затем создаем объект `learner`,

с которым мы взаимодействуем для обучения модели; мы не взаимодействуем с самой моделью, хотя и передаем его в качестве параметра. Мы также предоставляем параметр возврата (используем значение, предлагаемое в учебных материалах fast.ai):

```
learn = language_model_learner(data_lm, AWD_LSTM, drop_mult=0.3)
```

Когда у нас есть объект `learner`, мы можем найти оптимальную скорость обучения аналогично реализации в главе 4, за исключением того, что он встроен в библиотеку и использует экспоненциальное скользящее среднее для сглаживания графика, который в нашей реализации имеет довольно острые пики:

```
learn.lr_find()
learn.recorder.plot()
```

Из графика на рис. 9.12 видно, что `1e-2` — это то место, где кривая резко падает, поэтому мы выберем эту скорость для обучения. Библиотека fast.ai вызвала метод `fit_one_cycle`, который использует планировщик обучения за один цикл (подробнее см. Дополнительные источники) и очень высокую скорость для обучения модели за меньшее количество эпох.

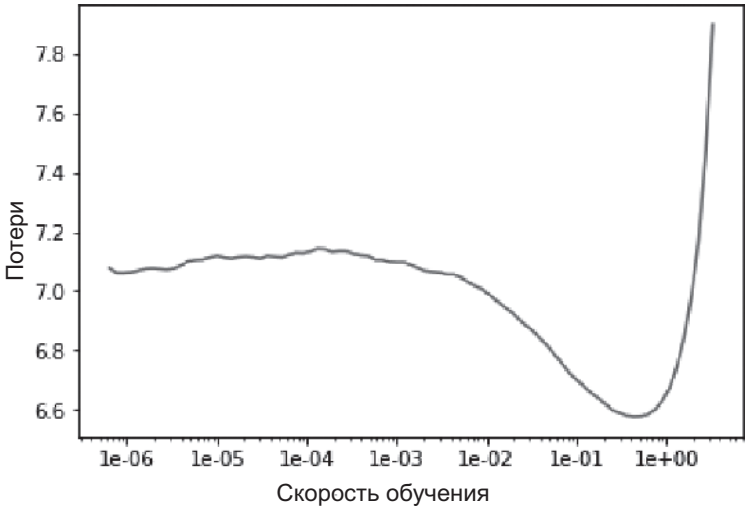

Рис. 9.12. График скорости обучения ULMFiT

Здесь мы обучаем модель только один цикл и сохраняем настроенную верхнюю часть сети (*кодировщик*):

```
learn.fit_one_cycle(1, 1e-2)
learn.save_encoder('twitter_encoder')
```

Завершив тонкую настройку языковой модели (вы можете поэкспериментировать с большим количеством циклов обучения), создаем новый `databunch` для актуальной задачи классификации:

```
twitter_classifier_bunch = TextList
           .from_csv("./twitter-data/",
           'train-processed.csv', cols=5,
           vocab=data_lm.vocab)
           .split_by_rand_pct()
           .label_from_df(cols=0)
           .databunch())
```

Единственное реальное отличие здесь состоит в том, что мы предоставляем фактические маркировки с помощью `label_from_df` и передаем объект `vocab` из обучения языковой модели, которое мы провели ранее, с тем чтобы они использовали то же преобразование слов в числа, а затем мы можем создать новый объект `text_classifier_learner`, где библиотека создает модели вместо вас. Теперь загружаем настроенный кодировщик в новую модель и начинаем процесс обучения снова:

```
learn = text_classifier_learner(data_clas, drop_mult=0.5)
learn.load_encoder('fine_tuned_enc')

learn.lr_find()
learn.recorder.plot()

learn.fit_one_cycle(1, 2e-2, moms=(0.8,0.7))
```

При небольшом количестве кода у нас есть классификатор, который говорит о точности в 76 %. Мы могли бы запросто улучшить этот показатель, обучая языковую модель большее количество циклов, добавляя дифференциальную скорость обучения и замораживая части модели во время обучения, и все это fast.ai поддерживает с помощью методов, определенных для объекта `learner`.

Что выбрать?

После этого небольшого экскурса в современные текстовые модели глубокого обучения у вас, вероятно, есть один вопрос: «Это все замечательно, но какую из них мне выбрать?» В целом, если вы работаете над проблемой классификации, я предлагаю начать с ULMFiT. BERT впечатляет, но ULMFiT конкурирует с BERT с точки зрения точности, и у нее есть дополнительное преимущество, заключающееся в том, что вам не нужно покупать огромное количество кредитов TPU, чтобы получить наилучший результат. Для большинства людей настройки ULMFiT с одним графическим процессором будет достаточно.

А что касается GPT-2, если вам нужен сгенерированный текст, то да, это лучшее, что можно использовать, но если вы хотите решать задачи классификации, то в этом случае вам будет сложнее получить производительность, схожую с ULMFiT или BERT. Есть кое-что, что, на мой взгляд, было интересно попробовать. Вы можете дать GPT-2 послабление при аугментации данных; если у вас есть такой набор данных, как Sentiment140, который мы использовали на протяжении всей этой книги, почему бы не настроить модель GPT-2 для них и не использовать ее для генерации большего количества данных?

Заключение

В этой главе я рассказал о богатом мире PyTorch, в том числе о библиотеках с имеющимися моделями, которые вы можете импортировать в свои собственные проекты, о некоторых передовых подходах к аугментации данных, которые можно применить к любому домену, а также о состязательных семплах, которые могут испортить день вашей модели, и о способах защиты от них. Я надеюсь, в конце нашего путешествия вы поймете, как появляются нейронные сети и как пропускать через них изображения, текст и аудио в виде тензоров. Теперь вы умеете обучать их, аугментировать данные, экспериментировать со скоростью обучения и даже отлаживать модели, если они работают не совсем правильно. А еще вы сможете упаковать их в Docker и сделать так, чтобы они обрабатывали запросы из внешнего мира.

Куда двигаться дальше? Подумайте об изучении форумов PyTorch и другой документации на сайте. Я определенно рекомендую посетить сообщество fast.ai, даже если вы не собираетесь использовать эту библиотеку; это дружелюбный к новичкам центр активности, наполненный хорошими идеями и людьми, экспериментирующими с новыми подходами!

Идти в ногу с последними достижениями в области глубокого обучения становится все сложнее.

Большинство статей публикуются на arXiv (https://arxiv.org/), но их количество, похоже, растет почти в геометрической прогрессии; когда я писал это заключение, был выпущен XLNet (https://arxiv.org/abs/1906.08237), который явно превосходит BERT в решении различных задач. Это никогда не заканчивается! Чтобы помочь вам, я перечислил несколько аккаунтов в Twitter, где авторы часто советуют интересные статьи. Предлагаю подписаться на них, чтобы быть в курсе современных и интересных работ, и тогда вы, возможно, сможете использовать такой инструмент, как rXiv Sanity Preserver (http://arxiv-sanity.com/), чтобы впитывать еще больше информации, когда вы почувствуете себя более комфортно, погрузившись в мир глубокого обучения.

Напоследок я обучил модель GPT-2 на книге, и она хотела бы сказать несколько слов:

> Глубокое обучение *является ключевой движущей силой того, как мы работаем с современными приложениями глубокого обучения, и в дальнейшем глубокое обучение продолжит разрастаться в новые области, такие как классификация на основе анализа изображений, и в 2016 году NVIDIA представила архитектуру CUDA LSTM. Архитектура LSTM стала не только более распространенным, но и более дешевым и простым в реализации способом для исследовательских целей, а CUDA оказалась очень конкурентоспособной архитектурой на рынке глубокого обучения.*

К счастью, вы видите, что существует еще множество вариантов развития, прежде чем мы, авторы, останемся без работы. Но, возможно, вы сможете изменить это!

Дополнительные источники

- Исследования современных методик сверхвысокого разрешения, https://arxiv.org/pdf/1902.06068.pdf
- Лекция Яна Гудфеллоу о GAN, https://www.youtube.com/watch?v=Z6rxFNMGdn0
- You Only Look Once (YOLO), семейство моделей быстрого обнаружения объектов с увлекательными статьями, https://pjreddie.com/darknet/yolo
- CleverHans, библиотека методов состязательной генерации для TensorFlow и PyTorch, https://github.com/tensorflow/cleverhans
- The Illustrated Transformer, всеобъемлющее путешествие по архитектуре Transformer, http://jalammar.github.io/illustrated-transformer

Некоторые интересные аккаунты в Twitter:

- @jeremyphoward — сооснователь fast.ai
- @miles_brundage — исследователь политик в OpenAI
- @BrundageBot — Twitter-бот, который генерирует ежедневную сводку интересных статей из arXiv (осторожно: публикует до 50 статей в день!)
- @pytorch — официальный аккаунт PyTorch

ОБ АВТОРЕ

Ян Пойнтер (Ian Pointer) — инженер data science, специализирующийся на решениях для машинного обучения (включая методы глубокого обучения) для нескольких клиентов из списка *Fortune 100*. В настоящее время Ян работает в *Lucidworks*, где занимается передовыми приложениями и разработкой NLP.

Он иммигрировал в США из Великобритании в 2011 году и стал американским гражданином в 2017-м.

ОБ ОБЛОЖКЕ

На обложке этой книги изображен красноголовый дятел (Melanerpes erythrocephalus). Эти птицы обитают в равнинных лесах Северной Америки и сосновых саваннах. Они мигрируют по всей восточной части США и Южной Канаде.

До взросления у красноголовых дятлов нет таких впечатляющих красных перьев. У взрослых особей черная спина и хвост, красная голова и шея и белые нижняя часть груди и брюхо. У молодых дятлов голова серого цвета. Вес взрослого дятла достигает 57–85 г, размах крыльев составляет 42 см, а длина туловища 18–23 см. Самки могут откладывать от четырех до семи яиц за раз. Они размножаются весной, давая до двух выводков за сезон. Самцы помогают высиживать яйца и кормить птенцов.

Красноголовые дятлы едят насекомых, которых они могут поймать в воздухе, семена, фрукты, ягоды и орехи. Они добывают пищу на деревьях и на земле характерным для них долблением поверхности. На зиму они прячут орехи в отверстия и щели в коре деревьев.

Многие из животных на обложках книг O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Иллюстрации на обложке выполнены Сьюзен Томпсон по черно-белой гравюре, взятой из *Pictorial Museum of Animated Nature*.

Ян Пойнтер

Программируем с PyTorch: Создание приложений глубокого обучения

Перевели с английского А. Попова

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*М. Колесников*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Муханова*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Соловьева*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2020.
Наименование: книжная продукция.
Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 26.03.20. Формат 70×90/16. Бумага офсетная.
Усл. п. л. 18,720. Тираж 1000. Заказ 0000.

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;*
e-mail: books@piter.com

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217;*
e-mail: kuznetsov@piter.com