TensorFlow.js

JavaScript для глубокого обучения

Франсуа Шолле
и
Эрик Нильсон
Стэн Байлесчи
Шэнкуинг Цэй

MANNING

Deep Learning with JavaScript

NEURAL NETWORKS IN TENSORFLOW.JS

SHANQING CAI
STANLEY BILESCHI
ERIC D. NIELSEN
WITH FRANÇOIS CHOLLET

FOREWORD BY NIKHIL THORAT
DANIEL SMILKOV

MANNING
SHELTER ISLAND

Франсуа Шолле
и Эрик Нильсон, Стэн Байлесчи, Шэнкуинг Цэй

JavaScript для глубокого обучения

TensorFlow.js

ПИТЕР®

Санкт-Петербург • Москва • Минск

2021

ББК 32.988.02-018
УДК 004.738.5
Д40

Франсуа Шолле, Эрик Нильсон, Стэн Байлесчи, Шэнкуинг Цэй

Д40 JavaScript для глубокого обучения: TensorFlow.js. — СПб.: Питер, 2021. — 576 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1697-3

Пора научиться использовать TensorFlow.js для построения моделей глубокого обучения, работающих непосредственно в браузере! Умные веб-приложения захватили мир, а реализовать их в браузере или серверной части позволяет TensorFlow.js. Данная библиотека блестяще портируется, ее модели работают везде, где работает JavaScript. Специалисты из Google Brain создали книгу, которая поможет решать реальные прикладные задачи. Вы не будете скучать над теорией, а сразу освоите базу глубокого обучения и познакомитесь с продвинутыми концепциями ИИ на примерах анализа текста, обработки речи, распознавания образов и самообучающегося игрового искусственного интеллекта.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1617296178 англ.
ISBN 978-5-4461-1697-3

Краткое содержание

Оглавление

Часть II. Введение в TensorFlow.js

Часть III. Продвинутые возможности глубокого обучения с TensorFlow.js

Часть IV. Резюме и заключительное слово

Приложения

Предисловие

Когда мы только начинали разработку библиотеки TensorFlow.js (TF.js), ранее называвшейся deeplearn.js, машинное обучение (machine learning, ML) выполнялось в основном на Python. А поскольку мы писали программы на JavaScript и активно использовали машинное обучение в команде Google Brain, то быстро заметили возможность связать обе сферы. Сегодня благодаря TF.js новому поколению разработчиков из обширного сообщества JavaScript доступно создание и развертывание моделей машинного обучения, а также новые классы локальных вычислений.

TF.js не существовала бы в нынешнем виде без Шэнкуинга, Стэна и Эрика. Их работа над Python TensorFlow, включая отладчик TensorFlow, немедленное выполнение кода, а также инфраструктуру сборки и тестирования, обеспечила им уникальные возможности для соединения миров Python и JavaScript. Еще в самом начале разработки их команда поняла, что необходима библиотека поверх deeplearn.js, которая бы включала высокоуровневые стандартные блоки для создания моделей машинного обучения. Шэнкуинг, Стэн и Эрик совместно с другими разработчиками придумали API слоев TF.js (TF.js Layers) для преобразования моделей Keras в JavaScript, резко расширив таким образом множество доступных моделей в экосистеме TF.js. А когда разработка TF.js Layers была завершена, библиотека TF.js была выпущена для широкой публики.

Стремясь выяснить настроения, затруднения и пожелания разработчиков программного обеспечения, Кэрри Цай и Филип Гао развернули соцопрос на сайте TF.js. Эта книга представляет собой непосредственную реакцию на резюме этого опроса: «В результате исследования мы выяснили, что разработчики хотели бы видеть фреймворки машинного обучения, способные не просто предоставить API, — им требовалось нечто более фундаментальное: помощь в понимании и применении самого машинного обучения»[1].

В книге «JavaScript для глубокого обучения» удачно сочетаются элементы теории глубокого обучения и реальные примеры на JavaScript с использованием TF.js. Это прекрасный источник информации как для JavaScript-разработчиков, у которых еще нет опыта машинного обучения или теоретического математического образования, так и для опытных пользователей ML, желающих расширить свою деятельность на экосистему JavaScript. Эта книга написана с ориентацией на образец *Deep Learning with Python*[2] — один из самых популярных учебников по прикладному машинному обучению, автором которого является создатель Keras, Франсуа Шолле.

---

[1] *Cai C., Guo P.* Software Developers Learning Machine Learning: Motivations, Hurdles and Desires // IEEE Symposium on Visual Languages and Human-Centric Computing, 2019.

[2] *Шолле Ф.* Глубокое обучение на Python. — СПб.: Питер, 2020.

Основанная на наработках Шолле, «JavaScript для глубокого обучения» замечательно демонстрирует уникальные свойства JavaScript: интерактивность, переносимость и возможность локальных вычислений. Она охватывает основные понятия машинного обучения, но не уклоняется и от обсуждения последних достижений ML, таких как перевод текста, генеративные модели и обучение с подкреплением. В издании вы найдете даже практические советы по внедрению моделей ML в реальные приложения от специалистов, обладающих обширным опытом использования машинного обучения на практике. Примеры в этой книге снабжены иллюстрациями уникальных преимуществ экосистемы JavaScript. Исходный код всех примеров открыт, так что вы можете экспериментировать с ним и создавать его ветки.

Читатели, которые хотели бы изучить ML и использовать JavaScript в качестве основного языка программирования, могут рассматривать эту книгу как заслуживающий доверия источник информации. Надеемся, вы сочтете изложенные в этой книге передовые идеи машинного обучения и JavaScript полезными, а свое путешествие по ML на JavaScript — плодотворным и увлекательным.

Нихил Торат и Дэниел Смилков,
создатели deeplearn.js и технические
руководители TensorFlow.js

Введение

Вероятно, важнейшее событие в современной истории технологий — резкий рост возможностей нейронных сетей, начавшийся в 2012 году, когда увеличение размеров маркированных наборов данных, вычислительных мощностей и алгоритмических инноваций совокупно достигло критической массы. После этого нейронные сети сделали вполне решаемыми невыполнимые ранее задачи и резко повысили точность решения многих других задач, переведя их из сугубо теоретической плоскости в сферу практического применения в таких областях, как распознавание речи, маркирование изображений, создание генеративных моделей и рекомендательных систем. Причем это далеко не все.

На этом фоне наша команда из Google Brain начала разработку TensorFlow.js. При запуске данного проекта многие считали глубокое обучение на JavaScript забавной новинкой, даже диковинкой, развлечением, подходящим лишь для определенных сценариев использования, а не чем-то серьезным. В то время как для Python уже существовало несколько испытанных фреймворков для глубокого обучения, обладавших широкими возможностями, ландшафт машинного обучения на JavaScript оставался раздробленным и неполным. Большинство немногочисленных доступных в те времена библиотек JavaScript поддерживали только развертывание заранее обученных на других языках моделей (чаще всего на Python). Диапазон поддерживаемых типов моделей в тех немногих, что поддерживали создание и обучение с нуля, был весьма ограничен. Очень странная ситуация, если учесть популярность JavaScript и его повсеместное использование как в клиентской, так и в серверной частях систем.

TensorFlow.js — первая полнофункциональная библиотека промышленного уровня, предназначенная для создания нейронных сетей на JavaScript. Спектр предоставляемых ею возможностей довольно широк. Во-первых, она поддерживает большой диапазон слоев нейронных сетей, подходящих для различных типов данных, от числовых до текстовых, от аудио- до видеоданных. Во-вторых, TensorFlow.js предоставляет API, позволяющие загружать предобученные модели для логического вывода, тонко настраивать их, а также создавать и обучать модели с нуля. В-третьих, она предоставляет как высокоуровневый API а-ля Keras для специалистов-практиков, предпочитающих использовать испытанные типы слоев, так и низкоуровневый API а-ля TensorFlow для желающих реализовать более новые типы алгоритмов. Наконец, библиотека спроектирована в расчете на работу в широком диапазоне сред и типов аппаратного обеспечения, включая браузеры, серверную часть (Node.js), мобильные среды (например, React Native и WeChat), а также настольные компьютеры (electron). Кроме всего прочего, библиотека TensorFlow.js является полноправной

составной частью экосистемы TensorFlow/Keras и обеспечивает согласованность API и двустороннюю совместимость «модель — формат» с библиотеками Python.

Наша книга сыграет роль гида в вашем долгом путешествии по этому многомерному пространству возможностей. Мы выбрали маршрут, пролегающий в основном по первому измерению этого пространства (задачам моделирования), но он дополнен экскурсиями вдоль прочих измерений. Начнем с относительно простой задачи предсказания чисел на основе других чисел (регрессия) и постепенно перейдем к более сложным задачам, например к предсказанию классов по изображениям и последовательностям. Закончим же наше путешествие захватывающими вопросами использования нейронных сетей для генерации новых изображений и обучения агентов принимать решения (обучение с подкреплением).

Мы рассчитывали, что книга послужит не только сборником рецептов по написанию кода на TensorFlow.js, но и вводным курсом по основам машинного обучения на «родном» языке веб-разработчиков — JavaScript. Сфера глубокого обучения развивается очень быстро. Мы убеждены, что можно досконально разобраться в машинном обучении без формального математического описания и благодаря полученным знаниям в дальнейшем поддерживать актуальное представление об эволюции этих методик.

Взяв в руки эту книгу, вы сделали первый шаг к тому, чтобы стать частью быстро растущего сообщества разработчиков машинного обучения на JavaScript, уже создавших множество важнейших приложений на стыке JavaScript и глубокого обучения. Мы искренне верим, что книга станет искрой, которая воспламенит ваше желание творить в этой сфере.

Шэнкуинг Цэй, Стэн Байлесчи
и Эрик Нильсон,
сентябрь 2019, Кембридж, Массачусетс

Благодарности

Своей структурой эта книга обязана известному творению Франсуа Шолле *Deep Learning with Python*. И хотя код был переписан на другом языке программирования и добавлено немало нового материала, связанного с экосистемой JavaScript, а также призванного отразить новые разработки в этой области, ни эта книга, ни весь высокоуровневый API TensorFlow.js не воплотился бы в реальность без новаторской работы Шолле по Keras.

Весь наш путь от начала до завершения этой книги и проработки всего соответствующего кода оказался приятным и плодотворным благодаря неоценимой поддержке наших соратников по команде TensorFlow.js из компании Google. Предварительная работа Дэниела Смилкова и Нихила Тората над низкоуровневыми ядрами WebGL и обратным распространением заложила надежный фундамент для создания и обучения моделей. Главным образом благодаря работе Ника Кригера над привязкой Node.js к библиотеке C TensorFlow мы теперь можем использовать один код для запуска нейронных сетей в браузере и Node.js. Без созданного Дэвидом Зоргелем и Кангли Чжаном API данных TensorFlow.js глава 6 этой книги не появилась бы на свет, а глава 7 стала возможной благодаря работе Янника Асогба в области визуализации. Описанные в главе 11 методики оптимизации были бы невозможны без работ Пинь Ю по интерфейсам операционного уровня (op-level) для TensorFlow. Наши примеры работали бы намного медленнее без оптимизации, специально проведенной Эн Юань. А в успехе книги в целом важнейшую роль сыграло руководство Сары Сирахуддин, Сандипа Гупта и Браджеша Кришнасвами.

Мы бы неизбежно сбились с пути истинного без поддержки и поощрения Д. Скалли, тщательно проверившего все главы. Мы также глубоко признательны за все слова ободрения от Фернанды Виегас, Мартина Ваттенберга, Хала Абельсона и многих других наших соратников из Google. Наша рукопись значительно улучшилась в результате подробного обзора, сделанного Франсуа Шолле, Нихилом Торатом, Дэниелом Смилковым, Джейми Смитом, Брайаном К. Ли и Аугустусом Оденой, как и благодаря обстоятельному обсуждению с Сухашем Шивакумаром.

Особенное удовольствие при создании такого проекта, как TensorFlow.js, состоит в возможности совместной работы и общения со всемирным сообществом разработчиков программного обеспечения с открытым исходным кодом. TensorFlow.js повезло, что над ним работали такие талантливые и вдохновенные разработчики, как Манраж Сингх, Каи Сасаки, Джош Гартмен, Саша Илларионов, Дэвид Сэндерс, syt123450@ и многие другие люди, чья неустанная работа над этой библиотекой не только расширила ее возможности, но и повысила качество кода. Манраж Сингх также выступил автором примера по обнаружению попыток фишинга из главы 3 данной книги.

Мы благодарны команде наших редакторов из издательства Manning Publications. Неустанная работа Брайана Сойера, Дженнифер Стаут, Ребекки Райнхарт и Мехмеда Пазика, а также многих других сотрудников позволила нам, авторам, сосредоточить свое внимание на написании рукописи. Марк-Филип Юж отвечал за всесторонний внимательный технический обзор материала книги. Особая благодарность нашим научным редакторам, рекомендации которых мы учли при подготовке издания. В их числе Ален Ломпо, Андреас Рефсгард, Бу Нгуен, Дэвид Ди Мариа, Эдин Капич, Эдвин Куок, Эоган О'Доннелл, Эван Уоллес, Джордж Томас, Джулиано Бертотти, Джейсон Хайлс, Марчо Николау, Майкл Уолл, Пауло Нуин, Пьетро Маффи, Полина Кесельман, Прабхати Пракаш, Райан Бэрроуз, Сатеш Саху, Суреш Рангараджулу, Урсин Стаус и Виджанат Рао.

Мы также благодарны читателям раннего издания (MEAP) книги, нашедшим немало типографских и технических ошибок и указавшим нам на них.

Наконец, все это было бы невозможно без безграничного понимания и жертв со стороны наших семей. Шэнкуинг Цэй хотел бы выразить глубочайшую признательность своей жене Вэй, а также своим родителям и родителям жены за их помощь и поддержку во время растянувшегося на год написания книги. Стэн Байлесчи хотел бы сказать спасибо своим матери и отцу, а также приемным родителям, благодаря начальной поддержке и наставлениям которых он сумел сделать успешную карьеру в области науки и инженерии. Он также хотел бы поблагодарить свою жену Констанцию за любовь и поддержку. Эрик Нильсон хотел бы сказать спасибо своим друзьям и семье.

Об этой книге

Для кого предназначено издание

Книга написана для тех программистов с практическими знаниями JavaScript и опытом разработки веб-клиентской части либо прикладной части на основе Node.js, которые хотели бы заняться глубоким обучением. Она будет полезна следующим двум группам читателей.

- JavaScript-программистам без особого опыта работы с машинным обучением и знания его математических основ, стремящимся хорошо разобраться в том, как функционирует глубокое обучение, и получить практические знания технологического процесса глубокого обучения, чтобы иметь возможность решать распространенные задачи науки о данных, такие как классификация и регрессия.
- Веб- и Node.js-разработчикам, перед которыми стоит задача развертывания предобученных моделей в своем веб-приложении или стеке прикладной части.

Для первой группы читателей в книге подробно «разжевываются» основные понятия машинного и глубокого обучения. Это делается на интересных примерах кода JavaScript, готовых для дальнейших экспериментов и исследований. Вместо формальных математических формул мы используем схемы, псевдокод и конкретные примеры, чтобы помочь вам усвоить на интуитивном уровне, но достаточно прочно принципы работы глубокого обучения.

Для второй группы читателей мы рассмотрим основные этапы преобразования уже существующих моделей (например, из библиотек обучения Python) в совместимый с веб или Node формат, подходящий для развертывания в клиентской части или стеке Node. Особое внимание мы уделяем практическим вопросам, таким как оптимизация размера и производительность модели, а также особенностям различных сред развертывания, от серверов до расширений браузеров и мобильных приложений.

Для всех читателей подробно описывается API TensorFlow.js для ввода, обработки и форматирования данных, для создания и загрузки моделей, а также для выполнения вывода, оценки и обучения.

Наконец, книга окажется полезной в качестве вводного руководства как по простым, так и по продвинутым нейронным сетям всем заинтересованным читателям с техническим складом ума, которым не приходится регулярно программировать на JavaScript или каком-либо другом языке.

Структура издания

Эта книга разбита на четыре части. В первой части, состоящей лишь из главы 1, вы познакомитесь с общим ландшафтом искусственного интеллекта, машинного и глубокого обучения, а также узнаете смысл реализации глубокого обучения на JavaScript.

Вторая часть представляет собой неспешное введение в наиболее базовые и часто встречающиеся понятия глубокого обучения.

- Главы 2 и 3 дают плавное введение в машинное обучение. Глава 2 посвящена простой задаче предсказания отдельного числа по набору чисел путем подбора прямой (линейная регрессия), на которой иллюстрируется работа процесса обратного распространения ошибки (основы глубокого обучения). В главе 3 идеи главы 2 расширяются: вам предстоит познакомиться с нелинейными, многослойными нейронными сетями и задачами классификации. Вы поймете, что такое нелинейности, как они функционируют и почему именно они обеспечивают подлинную мощь глубоким нейронным сетям.
- Глава 4 посвящена обработке данных изображений и архитектурам нейронных сетей, ориентированным на решение связанных с ними задач машинного обучения: сверточным нейронным сетям (convnets). Кроме того, на примере аудиоданных мы продемонстрируем, почему свертка — универсальный метод, применимый не только к изображениям.
- Глава 5 тоже посвящена сверточным нейронным сетям и подобным изображениям данным, но здесь в центре внимания оказывается перенос обучения: обучение новых моделей на основе уже существующих, вместо того чтобы начинать с чистого листа.

В части III приведен систематический обзор более сложных вопросов глубокого обучения для тех пользователей, кто хотел бы разобраться в самых современных технологиях. При этом делается упор на конкретные непростые сферы систем машинного обучения, а также на инструменты TensorFlow.js, предназначенные для работы с ними.

- В главе 6 обсуждаются методики работы с данными в контексте глубокого обучения.
- В главе 7 демонстрируются методики визуализации данных и модели для их обработки — важный и незаменимый этап любого технологического процесса глубокого обучения.
- В главе 8 мы сосредоточим внимание на важных вопросах недообучения и переобучения, а также на способах анализа и смягчения этих явлений. Во время обсуждения мы соберем все изученное до тех пор в книге в единую инструкцию, которую назовем «универсальным технологическим процессом машинного обучения». Эта глава заложит фундамент для передовых архитектур нейронных сетей и задач, обсуждаемых в главах 9–11.
- Глава 9 посвящена глубоким нейронным сетям, предназначенным для обработки последовательных и текстовых входных данных.

- В главах 10 и 11 обсуждаются более сложные темы глубокого обучения: генеративные модели (включая генеративные состязательные сети) и обучение с подкреплением соответственно.

В четвертой, заключительной части данной книги мы обсудим методики тестирования, оптимизации и развертывания моделей, обученных и преобразованных с помощью TensorFlow.js (глава 12), и в завершение книги резюмируем важнейшие понятия и технологические процессы (глава 13).

В конце вас ждут упражнения для проверки понимания пройденного материала и оттачивания на практике своих навыков глубокого обучения с помощью TensorFlow.js.

О коде

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и внутри обычного текста. В обоих случаях исходный код набран `таким моноширинным шрифтом`, чтобы его можно было отличить от обычного текста. Иногда код также набран **полужирным шрифтом**, чтобы подчеркнуть изменения по сравнению с предыдущими шагами в главе, например, при добавлении нового функционала к уже существующей строке кода.

Во многих случаях исходный код был переформатирован: мы добавили разрывы строк и переработали отступы, чтобы наилучшим образом использовать доступное место на страницах книги. В редких случаях этого оказывалось недостаточно и листинги включают маркеры продолжения строки (➥). Кроме того, из исходного кода нередко удалялись комментарии там, где код описывался в тексте. Многие листинги сопровождаются примечаниями к коду, подчеркивающими важные нюансы. Исходный код для всех листингов книги доступен для скачивания с GitHub по адресу https://github.com/tensorflow/tfjs-examples.

Дискуссионный форум книги

Покупка этого издания дает право на бесплатный доступ к частному веб-форуму издательства Manning, где можно оставлять свои комментарии о книге, задавать технические вопросы и получать помощь от авторов книги и других пользователей. Чтобы попасть на этот форум, перейдите по ссылке https://livebook.manning.com/#!/book/deep-learning-with-javascript/discussion. Узнать больше о форумах издательства Manning и правилах поведения на них можно на странице https://livebook.manning.com/#!/discussion.

Обязательства издательства Manning по отношению к своим читателям требуют предоставления места для содержательного диалога между читателями, а также между читателями и авторами. Эти обязательства не определяют какой-либо конкретный объем участия со стороны авторов, чей вклад в работу форума остается добровольным (и неоплачиваемым). Мы советуем вам задавать авторам интересные и трудные вопросы, чтобы их интерес не угас!

Об авторах

Шэнкуинг Цэй, Стэн Байлесчи и Эрик Нильсон — специалисты по разработке программного обеспечения из команды Google Brain, основные разработчики высокоуровневого API TensorFlow.js, в том числе примеров, документации и соответствующих утилит. Они использовали глубокое обучение на основе TensorFlow.js для реальных задач, например для альтернативных методов коммуникации с инвалидами. Все они имеют ученые степени от MIT.

Об иллюстрации на обложке

Рисунок на обложке озаглавлен *Finne Katschin* («Девушка-качинка»). Это иллюстрация костюма из книги Жака Грассе де Сан-Савье *Costumes de Différents Pays* («Наряды из разных стран»), опубликованной во Франции в 1797 году. Все иллюстрации прекрасно прорисованы и раскрашены вручную. Широкое разнообразие коллекции нарядов Грассе де Сан-Савье напоминает нам, насколько далеко отстояли друг от друга различные регионы мира всего 200 лет назад. Изолированные друг от друга люди говорили на различных диалектах и языках. На улицах городов и в деревнях по одной манере одеваться можно было легко понять, каким ремеслом занимается человек и каково его социальное положение.

Стили одежды с тех пор изменились, и столь богатое разнообразие различных регионов сгладилось. Зачастую непросто отличить даже жителя одного континента от жителя другого, не говоря уже о городах, регионах и странах. Возможно, мы пожертвовали культурным многообразием в пользу большего разнообразия личной жизни — и определенно более разносторонней и динамичной жизни технологической.

В наше время, когда книги по информационным технологиям так мало отличаются друг от друга, издательство Manning отдает должное изобретательности и инициативе компьютерного бизнеса обложками книг, основанными на богатом разнообразии местной жизни двухвековой давности, возвращенном к жизни иллюстрациями Жака Грассе де Сан-Савье.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Часть I

Актуальность и основные понятия

Часть I состоит из одной главы, по прочтении которой вы будете ориентироваться в основных понятиях, образующих контекст данной книги. В их числе искусственный интеллект, машинное обучение, глубокое обучение и взаимосвязи между ними. В главе 1 также рассказывается о пользе и возможностях глубокого обучения на JavaScript.

1 Глубокое обучение и JavaScript

В этой главе

- Что такое глубокое обучение; его связь с искусственным интеллектом и машинным обучением.
- Чем глубокое обучение выделяется среди разнообразных методик машинного обучения. Факторы, приведшие к нынешней революции глубокого обучения.
- Причины для глубокого обучения на JavaScript с помощью TensorFlow.js.
- Общая структура книги.

Шумиха вокруг искусственного интеллекта (ИИ) поднялась неспроста: революция глубокого обучения, как ее иногда называют, — реальность. Термин *«революция глубокого обучения»* характеризует стремительное увеличение скорости и количества методик глубоких нейронных сетей, начавшееся около 2012 года и продолжающееся по сей день. С тех пор глубокие нейронные сети стали применять для все более широкого диапазона задач, что позволило машинам в одних случаях решать доселе неразрешимые задачи, а в других — резко повышать степень безошибочности[1] (см. примеры в табл. 1.1). Эксперты в области ИИ были ошеломлены многими из этих прорывов в сфере нейронных сетей. А открывшиеся благодаря такому прогрессу возможности побудили специалистов, использующих нейронные сети, к действию.

[1] Здесь и далее в книге так переводится термин accuracy. — *Примеч. пер.*

JavaScript традиционно был языком, предназначенным для создания UI браузеров и прикладной бизнес-логики (с помощью Node.js). И те, кто привык выражать свои идеи и творить на JavaScript, чувствовали себя несколько обойденными революцией глубокого обучения, казавшейся исключительной прерогативой таких языков, как Python, R и C++. Цель книги — свести воедино глубокое обучение и JavaScript с помощью библиотеки TensorFlow.js, предназначенной для глубокого обучения на JavaScript. Мы делаем это, чтобы JavaScript-разработчикам вроде вас не нужно было учить новый язык для создания глубоких нейронных сетей; а главное, мы убеждены, что глубокое обучение и JavaScript просто созданы друг для друга.

Подобное взаимообогащение открывает уникальные возможности, недоступные ни в каком другом языке программирования. Причем как для JavaScript, так и для глубокого обучения. Благодаря JavaScript приложения глубокого обучения могут запускаться на большем количестве платформ, охватывая бóльшую аудиторию, а также становиться более наглядными и интерактивными. Благодаря глубокому обучению создаваемые JavaScript-разработчиками приложения становятся более «интеллектуальными». Далее в этой главе мы опишем, как именно это происходит.

В табл. 1.1 перечислены некоторые из наиболее захватывающих достижений глубокого обучения, появившихся в ходе вышеупомянутой революции. В книге мы выбрали несколько таких приложений и создали примеры реализации их на TensorFlow либо в полном объеме, либо в урезанном варианте. Эти примеры будут подробно описаны в следующих главах. То есть мы не остановимся на том, что будем молча восхищаться технологическими прорывами: мы изучим их, разберемся в них и реализуем их все на JavaScript.

Таблица 1.1. Примеры задач, безошибочность которых существенно выросла благодаря методикам глубокого обучения с момента старта революции глубокого обучения в 2012 году. Этот список, конечно, отнюдь не исчерпывающий. Прогресс, несомненно, будет продолжаться в ближайшие месяцы и годы

Задача машинного обучения	**Соответствующая технология глубокого обучения**	**Где в данной книге мы решали аналогичную задачу с помощью TensorFlow.js**
Категоризация содержимого изображений	Глубокие сверточные нейронные сети, например ResNet[1] и Inception[2], позволили снизить частоту ошибок при классификации изображений в ImageNet с ~25 % в 2011 году до менее чем 5 % в 2017 году[3]	Обучение сверточных нейронных сетей для MNIST (см. главу 4); перенос обучения и вывод с помощью MobileNet (см. главу 5)

Продолжение ⇩

[1] *He K. et al.* Deep Residual Learning for Image Recognition // Proc. IEEE Conference Computer Vision and Pattern Recognition (CVPR), 2016. Pp. 770–778. http://mng.bz/PO5P.

[2] *Szegedy C. et al.* Going Deeper with Convolutions // Proc. IEEE Conference Computer Vision and Pattern Recognition (CVPR), 2015. Pp. 1–9. http://mng.bz/JzGv.

[3] Результаты конкурса по масштабному распознаванию визуальных образов (Large Scale Visual Recognition Challenge, 2017, ILSVRC2017). http://image-net.org/challenges/LSVRC/2017/results.

Таблица 1.1 *(продолжение)*

Задача машинного обучения	Соответствующая технология глубокого обучения	Где в данной книге мы решали аналогичную задачу с помощью TensorFlow.js
Нахождение объектов в изображениях	Различные варианты глубоких сверточных нейронных сетей[1] позволили снизить ошибку нахождения с 0,33 в 2012 году до 0,06 в 2017 году	YOLO в TensorFlow.js (см. раздел 5.2)
Перевод с одного естественного языка на другой	Нейронный машинный перевод Google (GMNT) снизил ошибку перевода примерно на 60 % по сравнению с лучшими обычными методиками машинного перевода[2]	Модели преобразования последовательностей в последовательности с долгой краткосрочной памятью (LSTM) и механизмом внимания (см. главу 9)
Распознавание непрерывной речи с обширным словарным запасом	С помощью основанной на LSTM архитектуры «кодировщик — внимание — декодировщик» можно добиться более низкого отношения количества слов к количеству ошибок, чем при использовании лучших систем распознавания речи без глубокого обучения[3]	Распознавание непрерывной речи с ограниченным словарным запасом (см. главу 9)
Генерация правдоподобно выглядящих изображений	Генеративные состязательные сети (GAN) теперь способны генерировать правдоподобно выглядящие изображения на основе обучающих данных (см. https://github.com/junyanz/CycleGAN)	Генерация изображений с помощью вариационных автокодировщиков (VAE) и GAN (см. главу 9)
Генерация музыки	Рекуррентные нейронные сети (RNN) и VAE помогают в создании музыки к фильмам и новых звуков музыкальных инструментов (см. https://magenta.tensorflow.org/demos)	Обучаем LSTM генерировать текст (см. главу 9)
Обучение игре в игры	Глубокое обучение плюс обучение с подкреплением (RL) позволяют машинам играть в простые игры Atari с одними пикселами в качестве входных данных[4]. А благодаря сочетанию глубокого обучения и алгоритма поиска по дереву методом Монте-Карло программа AlphaZero достигла поистине сверхчеловеческого уровня игры в го, просто играя сама с собой[5]	Использование RL для решения задачи управления типа «тележка с шестом» и реализации компьютерной игры «Змейка» (см. главу 11)

1 *Chen Y. et al.* Dual Path Networks. https://arxiv.org/pdf/1707.01629.pdf.

2 *Wu Y. et al.* Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation // submitted 26 Sept. 2016. https://arxiv.org/abs/1609.08144.

3 *Chiu C.-C. et al.* State-of-the-Art Speech Recognition with Sequence-to-Sequence Models // submitted 5 Dec. 2017. https://arxiv.org/abs/1712.01769.

4 *Mnih V. et al.* Playing Atari with Deep Reinforcement Learning // NIPS Deep Learning Workshop 2013. https://arxiv.org/abs/1312.5602.

5 *Silver D. et al.* Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm // submitted 5 Dec. 2017. https://arxiv.org/abs/1712.01815.

Задача машинного обучения	Соответствующая технология глубокого обучения	Где в данной книге мы решали аналогичную задачу с помощью TensorFlow.js
Диагностика заболеваний на основе медицинских снимков	Глубокие сверточные нейронные сети смогли достичь высокой специфичности и чувствительности, что можно было сравнить с работой врачей-офтальмологов при диагностике диабетической ретинопатии на основе снимков сетчатки пациентов[1]	Перенос обучения с помощью предварительно обученной модели изображений MobileNet (см. главу 5)

Но прежде, чем погрузиться в эти восхитительные практические примеры глубокого обучения, необходимо разобраться с общим контекстом ИИ, глубокого обучения и нейронных сетей.

1.1. Искусственный интеллект, машинное обучение, нейронные сети и глубокое обучение

Словосочетания *«искусственный интеллект»*, *«машинное обучение»*, *«нейронные сети»* и *«глубокое обучение»* обозначают близкие, но все же различные понятия. Чтобы ориентироваться в ошеломляющем мире ИИ, необходимо понимать, что эти термины обозначают. Далее мы приведем их определения и расскажем, как они связаны между собой.

1.1.1. Искусственный интеллект

Как видно из диаграммы Венна на рис. 1.1, ИИ — обширная область. Краткое определение этой диаграммы можно дать в следующем виде: попытка автоматизировать требующие интеллекта задачи, обычно выполняемые людьми. Соответственно, ИИ включает в себя машинное обучение, нейронные сети и глубокое обучение, а также многие подходы, отличные от машинного обучения. Например, первые шахматные программы содержали жестко «зашитые» правила, подобранные программистами. Их нельзя отнести к машинному обучению, поскольку машины явным образом программировались на решение задачи, а не подбирали стратегии решения задач путем обучения на данных. Долгое время многие специалисты считали, что для создания ИИ, мыслящего на уровне человека, достаточно написать большой набор явных правил оперирования знаниями и принятия решений. Этот подход — преобладающая

[1] *Gulshan V. et al.* Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs // JAMA. Vol. 316. No. 22, 2016. Pp. 2402–2410. http://mng.bz/wlDQ.

парадигма в сфере ИИ с 1950-х до конца 1980-х годов — известен под названием *«символический ИИ»* (symbolic AI)[1].

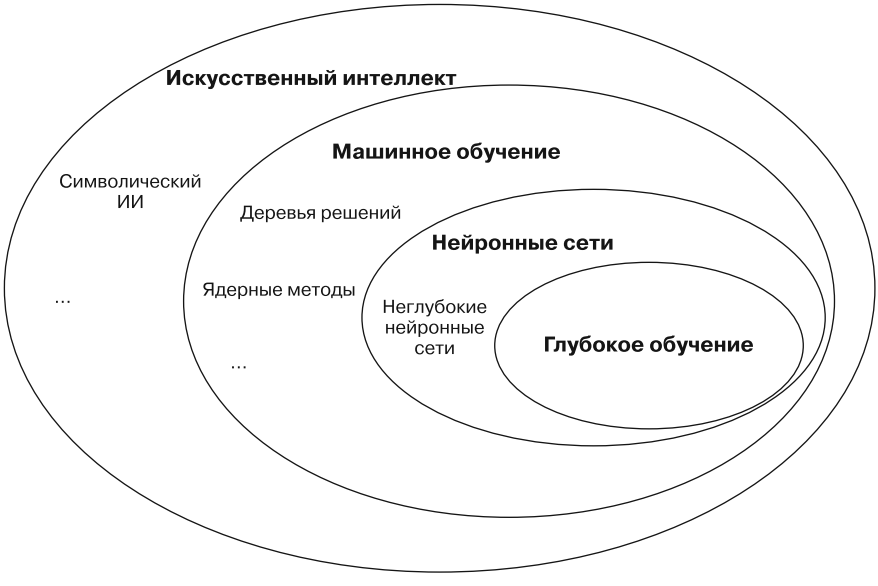


Рис. 1.1. Отношения между ИИ, машинным обучением, нейронными сетями и глубоким обучением. Как демонстрирует эта диаграмма Венна, машинное обучение является подобластью ИИ. В некоторых областях ИИ применяются отличные от машинного обучения подходы, например символический ИИ. Нейронные сети — подобласть машинного обучения. Существуют методики машинного обучения, не использующие нейронных сетей, например деревья принятия решений. Глубокое обучение представляет собой науку и искусство создания и применения глубоких нейронных сетей, которые содержат несколько слоев

1.1.2. Машинное обучение: отличия от традиционного программирования

В основе машинного обучения как отличной от символического ИИ подобласти ИИ лежит вопрос: может ли компьютер выйти за рамки задаваемых программистом правил поведения и самостоятельно найти способ решения конкретной задачи? Как вы видите, подход машинного обучения принципиально отличается от подхода, используемого в символическом ИИ. Если символический ИИ полагается на жестко «зашитые» знания и правила, то машинное обучение стремится избежать подобного.

---

[1] Одна из важных разновидностей символического ИИ — экспертные системы (expert systems). Узнать об этом больше можно в следующей статье Британской энциклопедии: http://mng.bz/7zmy (или в «Википедии» по адресу https://ru.wikipedia.org/wiki/Экспертная_система)).

Но если машине не даются явные указания, как выполнить задачу, то как же она может научиться это делать? Ответ: на основе примеров данных.

Эта идея открывает путь к совершенно новой парадигме программирования (рис. 1.2). В качестве примера парадигмы машинного обучения представьте себе, что вы работаете над приложением, обрабатывающим загруженные пользователями фотографии. Приложению требуется функциональность классификации фотографий на такие, где есть человеческие лица, и такие, где нет, поскольку обрабатывать их следует по-разному. Для этого вы хотите создать подпрограмму, получающую на входе произвольное изображение (состоящее из массива пикселов) и возвращающую двоичный ответ (фото с лицами/без лиц).

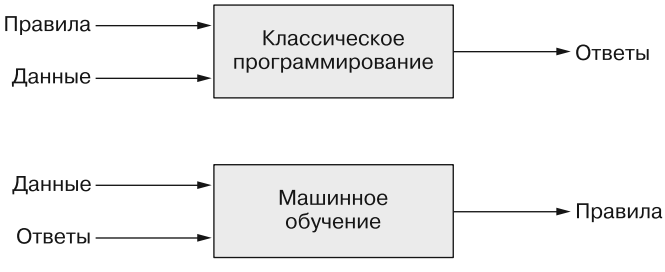


Рис. 1.2. Сравнение классической парадигмы программирования и парадигмы машинного обучения

Человек может выполнить подобное задание за долю секунды: способными на это нас делают мозг и жизненный опыт. Любому же программисту, каким бы опытным он ни был, будет очень непросто написать явный набор правил на каком-нибудь языке программирования (единственный доступный людям способ общения с компьютером) для точного определения, содержатся ли в заданном изображении лица людей. Вы можете потратить много дней, размышляя над кодом, который будет выявлять эллиптические контуры, напоминающие лица, глаза и рты, путем сложных арифметических действий над RGB-значениями пикселов, а также над написанием эвристических правил для геометрических связей между этими контурами. Но очень скоро вы осознаете, что при подобной работе придется произвольно выбирать логику и параметры без какого-либо понятного логического обоснования. Но главное: сложно добиться хорошей работы подобной системы![1] Никакой эвристический метод, который вы только можете себе представить, не способен справиться с мириадами вариантов лиц на реальных фотографиях, включая различия в размере, форме и чертах лица, выражении лица, прическе, цвете кожи, направлении, учитывая, возможно, частичную закрытость, наличие/отсутствие очков, условия освещения, наличие различных объектов на фоне и т. д.

[1] Попытки применения таких подходов уже были, и ничего хорошего из этого не вышло. В следующей обзорной статье приводятся хорошие примеры правил распознавания лиц, созданных вручную еще до наступления эпохи глубокого обучения: *Hjelmås E., Low B. K.* Face Detection: A Survey // Computer Vision and Image Understanding. Sept. 2001. Pp. 236–274. http://mng.bz/m4d2.

В парадигме машинного обучения создание подобного набора правил вручную считается напрасной тратой сил. Вместо этого берется набор изображений, одна часть из которых содержит лица, а другая — нет. А затем указывается желаемый (то есть правильный) ответ (наличие/отсутствие лица) для каждого изображения. Эти ответы называются *метками* (labels). Данная задача намного проще (фактически тривиальна). Конечно, если изображений много, маркирование их всех может занять некоторое время, но задачу маркирования можно разделить между несколькими людьми и выполнять параллельно. По завершении маркирования изображений можно применить алгоритм машинного обучения и дать возможность машине самой определить набор правил. При использовании правильной методики машинного обучения в результате получится обученный набор правил, позволяющий решать задачу обнаружения лиц с безошибочностью более 99 % — намного лучше, чем можно надеяться в случае правил, создаваемых вручную.

Из предыдущего примера видно, что машинное обучение представляет собой процесс автоматизации поиска правил для решения сложных задач. Такая автоматизация удобна для задач наподобие обнаружения лиц, в которых люди интуитивно чувствуют нужные правила и могут с легкостью маркировать данные. Для некоторых других задач правила не известны интуитивно. Например, рассмотрим задачу предсказания того, перейдет ли пользователь по отображаемому на веб-странице рекламному баннеру при известном содержимом страницы, баннера и прочей информации, скажем времени и местоположения. Ни один человек не может интуитивно давать точные предсказания для подобных задач. А даже если бы кто-то мог, паттерны, вероятно, будут меняться со временем и по мере появления нового контента и новых рекламных объявлений. Но маркированные обучающие данные можно найти в истории рекламного сервиса, они доступны в журналах серверов рекламы. А наличие данных и меток само по себе означает, что машинное обучение хорошо подходит для подобных задач.

На рис. 1.3 мы подробнее рассмотрим этапы машинного обучения. Существует два важных этапа. Первый называется *этапом обучения* (training phase). Здесь алгоритм получает данные и правильные ответы, которые вместе называют *обучающими данными* (training data). Отдельные пары из входных данных и желаемого ответа называются *примерами данных* или *выборками* (examples). На основе примеров данных в результате процесса обучения получаются автоматически подобранные *правила* (rules). И хотя правила были подобраны автоматически, нельзя сказать, что они подбираются с чистого листа. Другими словами, алгоритмы машинного обучения не проявляют творческих навыков при создании правил. В частности, специалист-человек намечает эскиз правил на отправном этапе обучения. Этот эскиз отражается в *модели* (model), формирующей *пространство гипотез* (hypothesis space) всех правил, которые только может усвоить машина. Без этого пространства гипотез пространство возможных правил оказалось бы ничем не ограниченным и бесконечным, а значит, неподходящим для поиска хороших правил за определенный промежуток времени. Мы во всех подробностях опишем возможные виды моделей и расскажем, как выбрать оптимальную для конкретной задачи. Пока же достаточно отметить, что в контексте глубокого обучения модели различаются количеством слоев, из которых состоит нейронная сеть, типами этих слоев и тем, как они состыковываются.

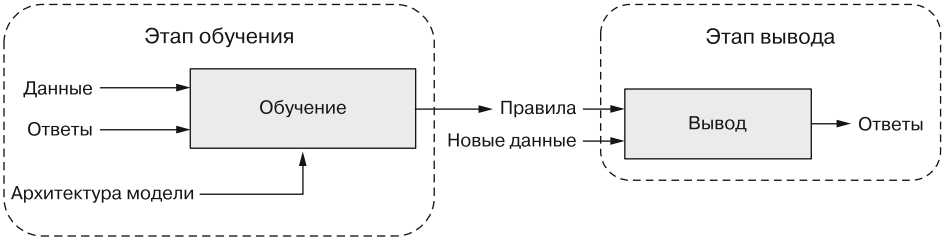


Рис. 1.3. Более подробная, по сравнению с рис. 1.2, схема парадигмы машинного обучения. Технологический процесс машинного обучения включает два этапа: обучение и вывод. Обучение представляет собой процесс автоматического подбора машиной правил преобразования данных в ответы. Результаты этапа обучения — усвоенные правила, заключенные в обученной модели, — образуют фундамент для этапа вывода. Вывод означает получение ответов для новых данных на основе обученной модели

На основе обучающих данных и архитектуры модели в результате процесса обучения формируются усвоенные правила. Они заключаются в обученной модели. Этот процесс изменяет (или уточняет) эскиз модели так, чтобы выходные данные модели становились все ближе и ближе к желаемым. Этап обучения может занимать от миллисекунд до дней, в зависимости от объема обучающих данных, сложности архитектуры модели и быстродействия аппаратного обеспечения. Такое направление машинного обучения — использование маркированных примеров для постепенного снижения погрешности выходных данных модели — называется *обучением с учителем* (supervised learning)[1]. Бóльшая часть описанных в книге алгоритмов глубокого обучения относится к обучению с учителем. После обучения модели усвоенные правила можно применять к новым данным — тем, которые не участвовали в процессе обучения. Это второй этап, *этап вывода*. Он требует меньшего объема вычислений, чем этап обучения, поскольку: 1) вывод обычно производится с одним элементом входных данных (например, одним изображением) за раз, в то время как обучение предполагает обработку всех обучающих данных; 2) во время вывода не требуется вносить изменения в модель.

Усвоение представлений данных

Машинное обучение связано с усвоением информации о данных. Но *какая именно* информация усваивается в ходе обучения? Ответ: способ эффективного преобразования данных или, другими словами, превращения старых представлений данных в новое, приближающее нас к решению текущей задачи.

Прежде чем продолжать, разберемся, что такое представление. По существу, это точка зрения на данные. На одни и те же данные можно взглянуть с разных сторон, что ведет к разным их представлениям. Например, цветное изображение может за-

[1] Другое направление машинного обучения — обучение без учителя (unsupervised learning), при котором используются немаркированные данные. Примерами обучения без учителя могут служить кластеризация (clustering, поиск отдельных подмножеств примеров в наборе данных) и обнаружение аномалий (anomaly detection, выяснение, достаточно ли сильно отличается конкретный пример данных от примеров из обучающего набора).

писываться в кодировках RGB или HSV (hue — saturation — value — «оттенок — насыщенность — значение»). Здесь слова *«кодировка»* (encoding) и *«представление»* (representation), по существу, означают одно и то же и могут заменять друг друга. При кодировании в этих двух форматах соответствующие пикселам числовые значения совершенно различны, хотя и соответствуют одному изображению. Различные представления удобны для решения разных задач. Например, чтобы найти все красные составляющие изображения, удобнее использовать представление RGB; а чтобы найти насыщенные цветом части того же изображения, лучше подойдет представление HSV. В этом вся суть машинного обучения: найти соответствующее преобразование для превращения старого представления входных данных в новое — подходящее для решения конкретной задачи, например определения местоположения машин в изображении или выяснения, есть ли на фотографии кошка и собака.

В качестве наглядного примера рассмотрим задачу, в которой на плоскости есть набор белых точек и несколько черных (рис. 1.4). Пусть нам нужно разработать алгоритм, который по заданным двумерным координатам (x, y) точки предсказывает, будет она белой или черной. В данном случае:

- роль входных данных играют двумерные декартовы координаты (x и y) точки;
- роль выходных данных играет предсказанный цвет точки (белый или черный).

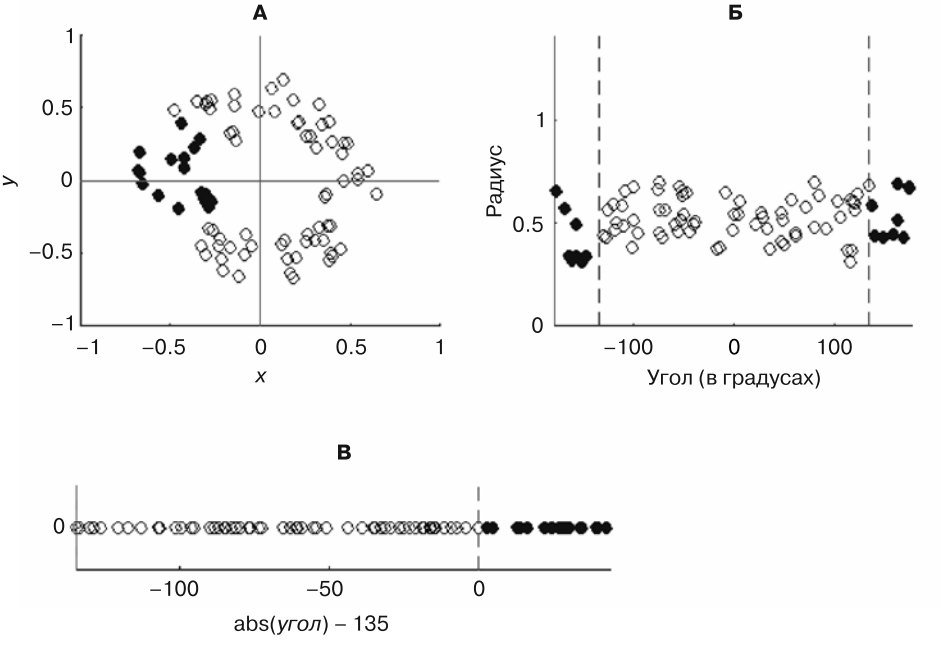


Рис. 1.4. Модельный пример преобразования представлений — самой сущности машинного обучения. Блок А: исходное представление набора данных, состоящего из белых и черных точек на плоскости. Блоки Б и В: после двух последовательных шагов преобразования исходное представление превращается в более подходящее для задачи классификации по цвету

Данные демонстрируют определенную закономерность в блоке А на рис. 1.4. Как может машина определить цвет точки по заданным координатам x и y? Просто сравнить x с числом не получится, ведь диапазон x-координат белых точек пересекается с диапазоном x-координат черных! Аналогично алгоритм не может воспользоваться для этой цели и y-координатами. Следовательно, становится ясно, что исходное представление плохо подходит для задачи классификации на белые и черные точки.

Нам нужно новое представление, на котором точки двух цветов разделялись бы более понятным образом. В данном случае мы преобразовали исходное декартово x-y-представление в представление в полярной системе координат. Другими словами, мы поставили каждой точке в соответствие: 1) ее угол — угол, сформированный осью X и прямой, соединяющей начало координат с этой точкой (см. пример в блоке А на рис. 1.4); 2) ее радиус — расстояние от начала координат. После этого преобразования получаем новое представление того же набора данных, как демонстрирует блок Б на рис. 1.4. Это представление лучше подходит для решения нашей задачи, поскольку диапазоны углов для черных и белых точек теперь вообще не пересекаются. Впрочем, это новое представление все еще неидеальное, поскольку не позволяет выполнить классификацию на точки белого и черного цветов, просто сравнивая угол с пороговым значением (например, с нулем).

К счастью, для этого можно применить еще одно преобразование, основанное на простой формуле:

```
(абсолютное значение угла) – 135 градусов
```

Полученное в результате представление, как видно из блока В, является одномерным. По сравнению с представлением из блока Б, в нем отброшена не относящаяся к делу информация о расстоянии от точек до начала координат. Но это представление идеально в том, что обеспечивает исключительно простой процесс принятия решения:

```
если значение < 0, то точка относится к белым;
   в противном случае точка относится к черным
```

В примере мы вручную задали двухшаговое преобразование представления данных. Вместо этого можно попытаться автоматически найти различные возможные преобразования координат на основе обратной связи о количестве правильно классифицированных точек. Это и будет машинным обучением. Число шагов преобразования при решении настоящих задач машинного обучения обычно намного превышает два, особенно при глубоком обучении, где может достигать сотен. Кроме того, в настоящем машинном обучении преобразования представлений обычно выглядят намного сложнее, по сравнению с показанными в этом простом примере. Непрерывные исследования в области глубокого обучения открывают все новые и все более перспективные преобразования. Но пример на рис. 1.4 демонстрирует саму суть поиска лучших преобразований. Это относится ко всем алгоритмам машинного обучения, включая нейронные сети, деревья принятия решений, ядерные методы и т. д.

1.1.3. Нейронные сети и глубокое обучение

Нейронные сети представляют собой подобласть машинного обучения, в которой преобразование представления данных выполняется системой, чья архитектура в какой-то мере отражает то, как соединяются нейроны в мозгу людей и животных. Как же нейроны соединяются друг с другом? У разных видов животных и в разных областях мозга по-разному. Но очень часто это происходит послойно. Многие части мозга млекопитающих организованы слоями, например кора головного мозга и кора мозжечка.

В общих чертах этот паттерн чем-то похож на общую организацию *искусственных нейронных сетей* (в мире вычислительной техники, где практически нет риска путаницы, они называются просто *нейронными сетями* (neural networks)). В них данные обрабатываются в несколько отдельных этапов, названных *слоями* (layers). Эти слои обычно размещаются друг поверх друга, причем соединены только непосредственно прилегающие друг к другу. На рис. 1.5 приведена простая (искусственная) нейронная сеть из четырех слоев. Входные данные (в данном случае изображение) поступают в первый слой (см. рис. 1.5, *слева*), затем последовательно проходят остальные слои. На каждом слое к представлению данных применяется новое преобразование. По мере движения данных по слоям представление все сильнее отличается от исходного и становится все ближе и ближе к поставленной перед нейронной сетью цели — а именно, к выдаче правильной метки для входного изображения. Последний слой (см. рис. 1.5, *справа*) выдает конечный результат работы нейронной сети, он же результат задачи классификации изображения.

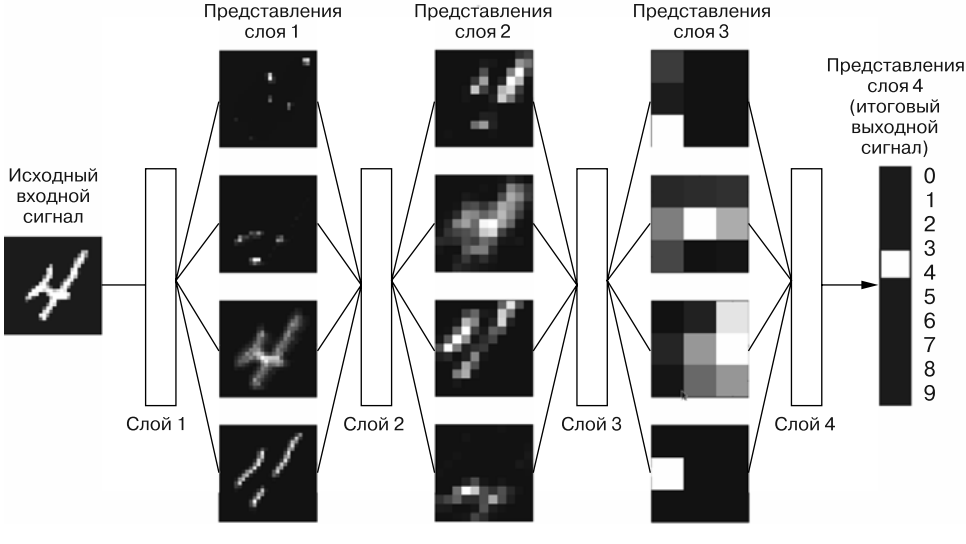


Рис. 1.5. Упрощенная схема нейронной сети, организованной послойно. Сеть классифицирует изображения рукописных цифр. Между слоями показаны промежуточные представления исходных данных. Воспроизводится с разрешения François Chollet, Deep Learning with Python, Manning Publications, 2017[1]

[1] *Шолле Ф.* Глубокое обучение на Python. — СПб.: Питер, 2020.

Слои нейронных сетей напоминают математические функции в том смысле, что представляют собой отображение входного значения на выходное. Впрочем, слои нейронных сетей отличаются от чисто математических функций тем, что обычно *сохраняют состояние*. Другими словами, они обладают внутренней памятью. Вместилищем памяти слоя служат его *веса* (weights). Что такое веса? Это просто набор связанных со слоем числовых значений, определяющих нюансы преобразования слоем входного представления в выходное. Например, часто применяемые *плотные* (dense) слои преобразуют входные данные, умножая их на матрицу и прибавляя к полученному результату вектор. Эти матрица и вектор — веса плотного слоя. При обучении нейронной сети на обучающих данных веса систематически меняются таким образом, чтобы минимизировать определенную величину, называемую *функцией потерь* (loss function), о которой мы поговорим подробнее на конкретных примерах в главах 2 и 3.

Хотя нейронные сети создавали под влиянием структуры мозга, не стоит слишком их очеловечивать. Цель нейронных сетей вовсе не в изучении или имитации работы мозга, это задача нейронауки — отдельной академической дисциплины. Нейронные сети призваны дать машинам возможность решать интересные практические задачи благодаря обучению на данных. Тот факт, что некоторые нейронные сети напоминают отдельные части живого мозга как структурно, так и функционально[1], действительно замечателен. Но обсуждение того, совпадение это или нет, выходит за рамки данной книги. В любом случае не стоит переоценивать это сходство. А главное, нет оснований утверждать, что обучение мозга происходит посредством какой-либо формы градиентного спуска — основного способа обучения нейронных сетей (мы рассмотрим его в следующей главе). Многие важные методики нейронных сетей, приведшие к революции глубокого обучения, были изобретены и приняты на вооружение не потому, что в их основе лежали нейронауки, а потому, что с их помощью нейронные сети быстрее и лучше решали реальные задачи.

Теперь, когда вы знаете, что такое нейронные сети, пора рассказать, что такое *глубокое обучение* (deep learning, DL). Глубокое обучение — это изучение и применение *глубоких нейронных сетей* (deep neural networks), представляющих собой попросту нейронные сети *с большим количеством слоев* (обычно от десятков до сотен). Слово *«глубокий»* здесь отражает большое число последовательных слоев представлений. Количество слоев, формирующих модель данных, называется *глубиной* (depth) модели. По-другому эту область изучения можно было бы назвать «многослойное усвоение представлений» или «иерархическое усвоение представлений». В современном глубоком обучении часто используются десятки или сотни последовательных слоев представлений — и обучение их всех происходит автоматически при взаимодействии с обучающими данными. Существуют и другие подходы к машинному обучению, в которых усваивается только один или два слоя представлений данных, соответственно они иногда называются *неглубоким обучением* (shallow learning).

Часто можно встретить ошибочное толкование слова «глубокий» в DL как относящегося к глубокому пониманию данных, то есть «глубокому» в смысле понимания

[1] Интересным примером функционального сходства могут служить входные сигналы, максимально активизирующие различные слои сверточной нейронной сети (см. главу 4), которые очень похожи на нейронные рецептивные поля различных частей человеческой зрительной системы.

предложений типа «свобода не дается бесплатно» или наслаждения противоречиями и ссылками на самого себя в картинах М. К. Эшера. Подобная «глубина» остается пока недостижимой целью для исследователей ИИ[1]. В будущем глубокое обучение может приблизить нас к подобной глубине, но ее явно не так просто достичь и оценить количественно, как добавить слои в нейронные сети.

ИНФОБОКС 1.1. Не только нейронные сети: другие популярные методики машинного обучения

Мы перешли от круга «Машинное обучение» на диаграмме Венна (см. рис. 1.1) непосредственно к внутреннему кругу «Нейронная сеть». Впрочем, имеет смысл вкратце упомянуть и не относящиеся к нейронным сетям методики машинного обучения: не только ради исторического контекста, но и потому, что некоторые из них могут встретиться вам в уже существующем коде.

Наивный байесовский классификатор (naive Bayes classifier) — одна из первых форм машинного обучения. Попросту говоря, теорема Байеса дает возможность оценки вероятности события по: 1) априорной вероятности этого события; 2) наблюдаемым фактам относительно этого события (называемым *признаками*). Пользуясь этой теоремой, можно классифицировать наблюдаемые точки данных по одной из множества категорий, выбрав наиболее вероятную (правдоподобную) в смысле наблюдаемых фактов категорию. В основе наивного байесовского классификатора лежит допущение, что наблюдаемые факты взаимно независимы (строгое и наивное допущение, отсюда и название).

Логистическая регрессия (logistic regression, logreg) — еще один метод классификации. Благодаря простоте и гибкости она до сих пор популярна, и исследователи данных зачастую начинают с нее работу над любой задачей классификации.

Ядерные методы (kernel methods), самый известный пример которых — *метод опорных векторов* (support vector machine, SVM), предназначены для решения задач бинарной (с двумя классами) классификации. Она выполняется путем отображения исходных данных в пространства более высокой размерности и поиска преобразования, которое максимизировало бы расстояние (так называемый *отступ* (margin)) между двумя классами примеров данных.

Деревья принятия решений (decision trees) — напоминающие блок-схемы структуры, с помощью которых можно классифицировать входные точки данных или предсказывать выходные значения по входным. На каждом шаге этой блок-схемы вы отвечаете на простой вопрос наподобие: «Превышает ли значение признака X заданное пороговое значение?» В зависимости от ответа (да/нет) выполняется переход к одному из двух других вопросов (тоже типа да/нет) и т. д. По достижении конца блок-схемы вы получаете окончательный ответ. Деревья принятия решений наглядны и удобны для интерпретации людьми.

Случайные леса (random forests) и *градиентный бустинг* (gradient-boosted machine) повышают безошибочность деревьев принятия решений за счет формирования ансамблей

[1] *Hofstadter D.* The Shallowness of Google Translate // The Atlantic. 30 Jan. 2018. http://mng.bz/5AE1.

из большого количества специализированных деревьев принятия отдельных решений. *Обучение на основе ансамблей* (ensembling, ensemble learning) — методика обучения набора (то есть ансамбля) отдельных моделей ML и использование во время вывода совокупности их выходных сигналов. На сегодняшний день градиентный бустинг — один из лучших (а то и лучший) алгоритмов для работы с несенсорной информацией (например, при обнаружении мошенничества с платежными картами). Наряду с глубоким обучением его чаще всего используют на конкурсах по науке о данных, например на конкурсах Kaggle.

Взлет, падение и новый взлет нейронных сетей, а также причины всего этого

Основные концепции нейронных сетей сформировались еще в 1950-х годах. Ключевые методики обучения нейронных сетей, в том числе метод обратного распространения ошибки, изобрели в 1980-х. Однако долгое время, с 1980-х до начала 2010-х годов, нейронные сети были незаслуженно забыты научным сообществом, частично вследствие популярности конкурирующих с ними методов, например SVM, а частично из-за отсутствия возможности обучать глубокие (многослойные) нейронные сети. Но в 2010 году сразу несколько исследователей, все еще занимавшихся нейронными сетями, осуществили несколько важных прорывов: группы исследователей под руководством Джеффри Хинтона из Университета Торонто, Йошуа Бенжио из Университета Монреаля, Ян Ле Куна из Нью-Йоркского университета, а также исследователи из швейцарского Института исследований в области ИИ Далле Молле (IDSIA). Они преодолели важные рубежи, в частности впервые практически реализовали глубокие нейронные сети на графических процессорах (GPU) и добились снижения частоты ошибок с примерно 25 % до менее чем 5 % в задаче машинного зрения ImageNet.

Начиная с 2012 года глубокие *сверточные нейронные сети* стали одним из лучших алгоритмов для всех задач машинного зрения (вообще говоря, они подходят для всех задач на восприятие). В числе прочих задач машинного зрения на восприятие можно назвать распознавание речи. На крупных конференциях по машинному зрению в 2015 и 2016 годах практически не встречалось докладов, в той или иной мере не связанных со сверточными нейронными сетями. В то же время глубокое обучение начали применять и для многих других типов задач, например обработки естественного языка. Нейронные сети полностью вытеснили SVM и деревья принятия решений из широкого диапазона приложений. Например, Европейская организация по вопросам ядерных исследований (ЦЕРН) на протяжении нескольких лет использовала методы на основе деревьев решений для анализа полученных от детектора ATLAS Большого адронного коллайдера данных по элементарным частицам. Но ЦЕРН постепенно перешел на использование глубоких нейронных сетей из-за большего быстродействия и простоты обучения на больших наборах данных.

Так что же выделяет глубокое обучение из шеренги существующих алгоритмов машинного обучения? (См. перечень некоторых популярных методов машинного

обучения, не связанных с нейронными сетями, в инфобоксе 1.1). Основная причина подобного успеха нейронных сетей заключается в более высоком их быстродействии для ряда задач. Но это не единственная причина. Глубокое обучение также упрощает решение задач, поскольку автоматизирует наиболее важный и трудный шаг технологического процесса машинного обучения: *проектирование признаков* (feature engineering).

Предшествовавшие нейронным сетям методы машинного обучения — неглубокое обучение — включали лишь преобразование входных данных в одно или два последовательных пространства представлений обычно путем простых преобразований наподобие многомерных нелинейных проекций (ядерные методы) или деревьев принятия решений. Но подобные методики чаще всего не позволяют получить изощренные представления, необходимые для сложных задач. Поэтому исследователям данных приходилось прикладывать немалые усилия, чтобы приспособить имеющиеся входные данные к обработке с помощью подобных методов: нужно было вручную проектировать подходящие слои представлений для своих данных. Этот процесс и называется *проектированием признаков*. Глубокое обучение же автоматизирует этот этап: с помощью DL можно усвоить все признаки за один проход, а не проектировать их самостоятельно. Благодаря этому технологические процессы машинного обучения существенно упрощаются, а хитроумные многоэтапные конвейеры зачастую заменяются одной простой сквозной моделью глубокого обучения. Благодаря автоматизации проектирования признаков глубокое обучение делает машинное обучение менее трудоемким и более ошибкоустойчивым, таким образом убивая одним выстрелом двух зайцев.

Глубокое обучение отличается двумя существенными нюансами того, как происходит усвоение данных. Во-первых, формирование все более сложных представлений выполняется инкрементно, послойно. Во-вторых, эти промежуточные инкрементные представления усваиваются совместно, каждый из слоев обновляется таким образом, чтобы учитывать потребности представления как уровнем выше, так и уровнем ниже. Благодаря этим двум свойствам глубокое обучение обрело намного бóльшую популярность, чем все предыдущие подходы к машинному обучению.

1.1.4. Почему глубокое обучение? И почему именно сейчас?

Если основные идеи и базовые методы нейронных сетей существовали еще в 1980-х, почему революция глубокого обучения началась только после 2012-го? Что поменялось за эти три десятилетия? В целом технический прогресс в области машинного обучения определяют три движущие силы:

- аппаратное обеспечение;
- наборы данных и тесты производительности;
- усовершенствование алгоритмов.

Рассмотрим эти факторы по очереди.

Аппаратное обеспечение

Глубокое обучение представляет собой прикладную науку, которая руководствуется скорее экспериментальными находками, а не теорией. Алгоритмы удалось усовершенствовать лишь тогда, когда появилось подходящее аппаратное обеспечение для проверки новых идей (или для масштабирования старых, как это часто бывает). Типичным моделям глубокого обучения, применяемым для машинного зрения или распознавания речи, необходимы на порядки большие вычислительные ресурсы, чем у обычного ноутбука.

В 2000-х годах такие компании, как NVIDIA и AMD, вложили миллиарды долларов в создание быстрых массово-параллельных микросхем (GPU) для работы все более фотореалистичных компьютерных игр. Это привело к появлению дешевых, узкоспециализированных суперкомпьютеров, предназначенных для визуализации на экране 3D-сцен в режиме реального времени. Эти инвестиции пригодились научному сообществу, когда в 2007 году компания NVIDIA создала универсальный интерфейс программирования линейки ее устройств — CUDA (сокращение от Compute Unified Device Architecture — архитектура унифицированного вычислительного устройства). Несколько GPU теперь могли заменить огромный кластер CPU в различных сильно распараллеливаемых задачах, начиная от моделирования физических процессов. Глубокие нейронные сети, включающие в основном большое количество матричных произведений и сложений, также относятся к сильно распараллеливаемым средствам.

В 2011 году некоторые исследователи — одними из первых были Дэн Чирешан и Алекс Крижевски — начали писать реализации нейронных сетей на CUDA. Сегодня вычислительная мощность высокопроизводительных GPU в параллельных вычислениях при обучении нейронных сетей в сотни раз больше, чем у обычного CPU. Без вычислительных возможностей современных GPU обучение многих передовых глубоких нейронных сетей было бы невозможно.

Данные и тесты производительности

Если аппаратное обеспечение и алгоритмы — паровой двигатель революции глубокого обучения, то данные — ее уголь: исходные материалы, служащие топливом для наших умных машин, без которых ничего не получилось бы. А что касается данных, то, помимо экспоненциального роста объемов аппаратного обеспечения для их хранения за последние 20 лет (согласно закону Мура), положение дел в этой области коренным образом изменил Интернет. Именно благодаря ему стало возможно собирать и распределенно хранить очень большие наборы данных для машинного обучения. Сегодня крупные компании работают с наборами данных изображений, видеоданных и естественного языка, которые без Интернета вообще невозможно было бы собрать. Например, настоящим кладом данных для машинного зрения стали задаваемые пользователями метки на Flickr. Аналогично — видео на YouTube. А «Википедия» — важнейший набор данных для обработки естественного языка.

Если и был один набор данных, послуживший катализатором революции глубокого обучения, то это, несомненно, ImageNet, состоящий из 1,4 миллиона изображений, разбитых на 1000 категорий. Набор данных ImageNet выделяет из прочих

не только его размер, но и соответствующий ежегодный конкурс. Как демонстрировали ImageNet и Kaggle, начиная с 2010 года проведение конкурсов — замечательный способ мотивировать исследователей и инженеров на открытие новых горизонтов. Общие тесты производительности, рекорды друг друга в которых стремились побить инженеры, сильно помогли текущему росту глубокого обучения.

Усовершенствование алгоритмов

До конца 2000-х годов отсутствовал надежный метод обучения очень глубоких нейронных сетей. В результате нейронные сети оставались довольно неглубокими, в них использовались только один или два слоя представлений, а значит, они ничем не выделялись из более утонченных неглубоких методов, например SVM и случайных лесов. Основная проблема была с градиентным распространением сигнала через глубокие эшелоны слоев. Используемый для обучения нейронных сетей сигнал обратной связи быстро угасал при увеличении количества слоев.

Все поменялось в районе 2009–2010 годов с появлением нескольких простых, но важных алгоритмических усовершенствований, улучшавших градиентное распространение сигнала, таких как:

- улучшенные функции активации для слоев нейронных сетей (например, выпрямленные линейные блоки (ReLU));
- более эффективные схемы для задания начальных значений весов (например, схема инициализации Глорота);
- улучшенные схемы оптимизации (например, RMSProp и ADAM).

Лишь когда благодаря этим усовершенствованиям стало возможно обучение моделей с десятью и более слоями, глубокое обучение показало, на что способно. Наконец, в 2014–2016 годах были открыты еще более продвинутые способы обратного распространения ошибки, в частности нормализация по мини-батчам, остаточные связи (residual connections) и разделяемые свертки с учетом глубины (depthwise separable convolutions). Сегодня можно обучать с нуля модели глубиной тысячи слоев.

1.2. Какой смысл в сочетании JavaScript и машинного обучения

Машинное обучение, как и другие сферы ИИ и науки о данных, обычно реализуется на традиционных языках программирования, ориентированных на прикладную часть, например Python и R, работающих на серверах или рабочих станциях вне браузера[1]. Подобное положение дел неудивительно. Обучение глубоких нейронных сетей часто требует вычислений с участием многих ядер процессора и GPU, недоступных непосредственно на вкладках браузера. Ввод и обработку колоссальных объемов

[1] *Deoras S.* Top 10 Programming Languages for Data Scientists to Learn in 2018 // Analytics India Magazine, 25 Jan. 2018. http://mng.bz/6wrD.

данных, порой необходимых для обучения подобных моделей, удобнее производить в прикладной части: например, из нативной файловой системы практически неограниченного размера. До недавних пор глубокое обучение на JavaScript многие считали диковинной новинкой. В этом разделе мы назовем причины, почему глубокое обучение в среде браузера с помощью JavaScript имеет смысл для многих видов приложений, а также расскажем, какие уникальные возможности дает сочетание браузера и мощи глубокого обучения, особенно с помощью TensorFlow.js.

Во-первых, по завершении обучения модели при ML ее необходимо где-то развернуть для предсказаний на реальных данных (например, классификации изображений и текста, поиска событий в потоках аудио- или видеоданных и т. д.). Без развертывания обучение модели — пустая трата вычислительных ресурсов. При этом нередко желательно или даже обязательно, чтобы это «где-то» было клиентской частью с веб-интерфейсом. Вероятно, вы понимаете важность браузеров в целом. На стационарных компьютерах и ноутбуках браузер — основное средство доступа к контенту и сервисам Интернета. Именно в них пользователи проводят бо́льшую часть времени. И именно с их помощью пользователи выполняют бо́льшую часть своей повседневной работы, связываются с друзьями и развлекаются. Широкий спектр работающих в браузере приложений обеспечивает обширные возможности машинного обучения на стороне клиента. Что касается мобильных клиентских частей, браузеры отстают от нативных мобильных приложений в смысле вовлеченности пользователей и проводимого времени. Но мобильные браузеры тем не менее заслуживают внимания из-за более широкого охвата аудитории, мгновенного доступа и более быстрого цикла разработки[1]. На самом деле многие мобильные приложения, например Twitter и Facebook, благодаря гибкости и простоте использования переходят на веб-представление с использованием JavaScript для определенных типов контента.

Благодаря более широкому охвату аудитории браузер — логичный выбор для развертывания моделей глубокого обучения, лишь бы в нем были доступны ожидаемые моделями виды данных. А какие виды данных доступны в браузерах? Очень многие! Возьмем, например, наиболее популярные приложения глубокого обучения: классификацию и обнаружение объектов в изображениях и видео, текстовую расшифровку речи, перевод с одного естественного языка на другой и анализ текстового контента. Браузеры оснащены, вероятно, наиболее исчерпывающим набором технологий и API для отображения (и, с разрешения пользователя, захвата) текстовых, визуальных, аудио- и видеоданных. В результате можно использовать многие впечатляющие модели глубокого обучения непосредственно в браузере, скажем, с помощью TensorFlow.js и простых средств преобразования. В последующих главах мы рассмотрим множество конкретных примеров развертывания моделей глубокого обучения в браузере. Например, после захвата изображений с веб-камеры можно запустить MobileNet с помощью TensorFlow.js для маркирования объектов, YOLO2 для отрисовки рамок вокруг обнаруженных объектов, Lipnet для чтения по губам или сеть CNN-LSTM для добавления названий изображений.

После захвата аудио с микрофона с помощью API WebAudio браузера можно запустить модели для распознавания устной речи в режиме реального времени,

[1] *Borde R.* Internet Time Spend in Mobile Apps, 2017–19: It's 8x than Mobile Web // DazeInfo, 12 Apr. 2017. http://mng.bz/omDr.

используя TensorFlow.js. Существуют подобные приложения и для текстовых данных, например анализирующие тональность высказываний в пользовательских текстах наподобие отзывов о фильмах (см. главу 9). Помимо этого, современные браузеры могут обращаться ко множеству датчиков мобильных устройств. Например, HTML5 предоставляет API для доступа к данным о географическом местоположении (широта и долгота), движении (ориентация и ускорение устройства) и внешней освещенности (см. https://mobilehtml5.org/). В сочетании с глубоким обучением и различными типами данных сведения от подобных датчиков открывают дорогу ко множеству новых приложений захвата.

У применения глубокого обучения в браузере есть пять дополнительных преимуществ: уменьшение затрат на серверы, снижение времени ожидания вывода, лучшая защита персональной информации, мгновенное ускорение вычислений с помощью GPU и моментальный доступ.

- *Затраты на серверы.* Это важный фактор при проектировании и масштабировании веб-сервисов. Для работы моделей глубокого обучения в отведенные сроки зачастую необходим значительный объем вычислений, и без ускорения вычислений с помощью GPU не обойтись. А если не развертывать модели в клиентской части, придется развертывать их на снабженных GPU машинах, например виртуальных машинах с поддерживающими CUDA GPU из Google Cloud или Amazon Web Services. Подобные облачные машины с GPU нередко обходятся недешево. Даже простейшая облачная машина с GPU сейчас стоит порядка 0,5–1 доллара в час (см. https://www.ec2instances.info/ и https://cloud.google.com/gpu). При росте трафика стоимость эксплуатации парка облачных машин с GPU тоже растет, не говоря уже о непростых задачах масштабирования и повышении сложности серверного стека. Всех этих проблем можно избежать при развертывании модели на клиенте. Накладные расходы на скачивание модели (составляющей обычно несколько мегабайт) на клиент можно снизить за счет кэширования в браузере и локального хранилища (см. главу 2).
- *Снижение времени ожидания вывода.* Для определенных типов приложений требования по времени ожидания настолько жестки, что модели глубокого обучения приходится запускать на стороне клиента. К этой категории относятся все приложения с обработкой аудио-, визуальных и видеоданных. Представьте себе, например, что будет, если для вывода потребуется передавать кадры изображений на сервер. Допустим, изображения захватываются с веб-камеры в довольно скромном разрешении 400×400 пикселов с тремя цветовыми каналами (RGB) 8-битной глубины со скоростью 10 кадров в секунду. Даже при использовании JPEG-сжатия размер каждого изображения будет около 150 Кбайт. В обычной мобильной сети с примерно 300-килобитной исходящей пропускной способностью загрузка одного такого изображения займет более 500 миллисекунд, что приведет к заметной и, возможно, неприемлемой для определенных приложений (например, игр) задержке. Расчеты не учитывают колебания (и, возможно, разрывы) связи в сети, дополнительное время для скачивания результатов вывода и немалый объем расхода мобильного трафика, а ведь каждый из этих факторов может оказаться фатальным.

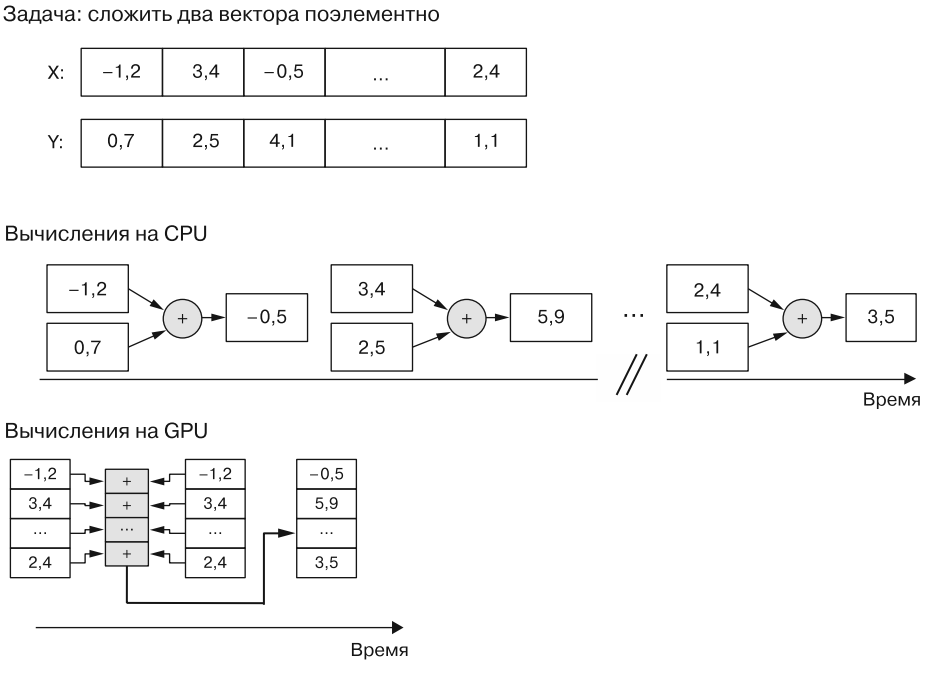
- Все эти проблемы с возможной задержкой и сетевыми соединениями решаются, если вывод выполнять на стороне клиента, ведь тогда данные не покидают устройства, на котором производятся вычисления. Работа приложений машинного обучения реального времени, например маркирование объектов и обнаружение людей на снимках с веб-камеры, возможна только при функционировании модели исключительно на клиенте. Даже для приложений, у которых нет жестких требований к задержке, снижение времени ожидания вывода модели приводит к повышению быстроты реакции, а значит, к более приятному опыту взаимодействия пользователя.
- *Защита персональных данных.* Еще одно преимущество того, что обучающие данные и данные вывода не покидают клиент. Вопрос защиты персональной информации обретает сейчас все больший вес. Для некоторых типов приложений защита персональной информации — обязательное требование. Очевидный пример — приложения, работающие с медицинскими и прочими связанными со здоровьем пользователей данными. Представьте себе приложение для диагностики кожных заболеваний, собирающее снимки кожи пациентов с веб-камеры и генерирующее возможные диагнозы с помощью глубокого обучения. Во многих странах нормы защиты персональной информации в сфере медицины запрещают перемещение подобных изображений для выполнения вывода на центральный сервер. А благодаря выводу данных модели в браузере передавать какие-либо сведения с пользовательского телефона и хранить их снаружи не требуется, что гарантирует защиту персональных медицинских данных пользователей.
- Представьте себе еще одно приложение на основе браузера, в котором применяется глубокое обучение для выдачи пользователям рекомендаций по улучшению написанного ими текста. Некоторые используют это приложение для подготовки важных материалов наподобие юридических документов и не хотели бы перемещения информации на удаленный сервер по Интернету. А выполнение модели исключительно в JavaScript браузера на стороне клиента — эффективный способ решения проблемы.
- *Мгновенное ускорение с помощью WebGL.* Помимо доступности данных, для работы моделей машинного обучения в браузере необходимы подходящие вычислительные мощности, которых обычно можно достичь с помощью GPU. Как уже упоминалось ранее, многие передовые модели глубокого обучения требуют таких вычислительных ресурсов, что без ускорения с помощью параллельных вычислений на GPU не обойтись (разве что пользователи готовы ждать по несколько минут одного-единственного результата вывода, что редкость в случае реальных приложений). К счастью, современные браузеры оснащены API WebGL, который хотя и предназначался изначально для ускорения визуализации двумерной и трехмерной графики, но может использоваться для ускорения работы нейронных сетей при соответствующем распараллеливании вычислений. Авторы TensorFlow.js заботливо подготовили библиотеку для ускорения вычислений глубокого обучения с помощью WebGL, так что для использования этих возможностей достаточно добавить одну строку импорта в JavaScript.

- Ускорение вычислений глубокого обучения с помощью WebGL, возможно, не всегда равноценно нативному, узкоспециализированному GPU-ускорению вычислений, например, с помощью CUDA и CuDNN от NVIDIA (используемому в библиотеках глубокого обучения на Python, в частности TensorFlow и PyTorch). Но оно все равно на порядки ускоряет расчеты нейронных сетей и обеспечивает вывод в режиме реального времени наподобие предлагаемых PoseNet возможностей выделения из изображений поз человеческого тела.
- А если уж вывод на основе предобученных моделей требует значительных вычислительных ресурсов, то обучение или перенос обучения на подобных моделях — тем более. Обучение и перенос обучения открывают путь к замечательным приложениям, например к персонализации моделей глубокого обучения, визуализации результатов глубокого обучения в клиентской части, а также федеративному обучению (созданию оптимальной модели за счет обучения одной и той же модели на нескольких устройствах с последующим агрегированием результатов). Ускорение TensorFlow.js с помощью WebGL позволяет обучать нейронные сети или точно подбирать их параметры с достаточной скоростью исключительно внутри браузера.
- *Мгновенный доступ* — вообще говоря, большой плюс работающих в браузере приложений в том, что их не нужно устанавливать: для запуска достаточно ввести URL или перейти по ссылке. Значит, можно избежать потенциально трудоемких и подверженных ошибкам шагов установки, а также возможных рисков контроля доступа при установке нового программного обеспечения. В контексте глубокого обучения в браузере ускорение нейронных сетей с помощью WebGL не требует специальных графических карт или установки драйверов для них, что часто бывает нетривиальной задачей. Большинство более или менее современных стационарных компьютеров, ноутбуков и мобильных устройств снабжены доступными для браузера и WebGL графическими картами. Подобные устройства сразу же готовы для вычислений нейронных сетей с WebGL-ускорением, достаточно лишь, чтобы на них был установлен TensorFlow.js-совместимый браузер. Это особенно привлекательно для тех случаев, где жизненно важна простота доступа — например, при изучении глубокого обучения в университетах.

ИНФОБОКС 1.2. Ускорение вычислений с помощью GPU и WebGL

Обучение моделей ML и последующий вывод возможны лишь при колоссальном количестве математических операций. Например, широко используемые «плотные» слои нейронных сетей требуют умножения большой матрицы на вектор и сложения результата с другим вектором. Типичная операция подобного плана означает тысячи или миллионы операций с плавающей точкой. Но у подобных операций зачастую есть важное свойство — они допускают *распараллеливание*. Например, сложение двух векторов можно разбить на множество меньших операций — сложений отдельных чисел. Причем эти меньшие операции не зависят друг от друга. Например, не нужно знать сумму двух элементов на нулевых позициях двух векторов, чтобы вычислить сумму

двух элементов на первых позициях. В результате можно выполнять эти меньшие операции одновременно, а не последовательно, вне зависимости от размера векторов. Последовательные вычисления, например CPU-нативная реализация сложения векторов, известны под названием SISD (Single Instruction Single Data — «один поток команд — один поток данных»). Параллельные вычисления на GPU известны под названием SIMD (Single Instruction Multiple Data — «один поток команд — несколько потоков данных»). Сложение отдельной пары чисел обычно занимает у CPU меньше времени, чем у GPU. Но общие затраты на вычисление такого большого количества операций намного меньше для SIMD GPU, чем для SISD CPU. Количество параметров глубоких нейронных сетей может достигать миллионов. Вычисления для конкретных входных данных могут включать миллиарды поэлементных математических операций (если не больше). Массово-параллельная обработка с помощью GPU в таких масштабах демонстрирует поистине потрясающие результаты.



Ускорение векторных операций на GPU по сравнению с CPU благодаря использованию WebGL возможностей параллельных вычислений на GPU

Если точнее, современные CPU тоже до какой-то степени способны выполнять инструкции SIMD. Впрочем, количество процессоров в GPU намного больше, чем в CPU (в сотни или тысячи раз), и они способны на выполнение инструкций на множестве срезов входных данных одновременно. Векторное сложение — относительно простая задача SIMD в том смысле, что на каждом шаге вычислений анализируется только одна позиция векторов, а результаты сложений на разных позициях не зависят друг от друга. Другие задачи SIMD, встречающиеся в машинном обучении, более сложны.

Например, при умножении матриц на каждом шаге вычислений используются данные из нескольких позиций и позиции взаимно зависимы. Но основная идея ускорения путем распараллеливания не меняется.

Интересно, что GPU изначально не были предназначены для ускорения вычислений нейронных сетей. Это видно даже из их названия: *графический процессор* (graphics processing unit). Основная задача GPU — обработка двумерной и трехмерной графики. Во многих графических приложениях, например в трехмерных компьютерных играх, очень важно, чтобы обработка производилась как можно быстрее и частота обновления кадров изображений была достаточно высокой для создания у пользователя приятного впечатления от игры. Именно с этой целью создатели GPU изначально применили распараллеливание типа SIMD. Но в качестве приятного бонуса подобные GPU с возможностью параллельных вычислений подходят и для нужд машинного обучения.

Библиотека WebGL, используемая TensorFlow.js для GPU-ускорения, изначально была создана для таких задач, как визуализация текстур (рельефов поверхностей) 3D-объектов в браузере. Но текстуры — это просто массивы чисел! А значит, можно притвориться, что эти числа — веса или функции активации нейронной сети, и перепрофилировать SIMD-операции библиотеки WebGL над текстурами для нейронных сетей. Именно так TensorFlow.js и выполняет ускорение вычислений нейронных сетей в браузере.

Помимо вышеупомянутых преимуществ, приложениям машинного обучения на основе браузеров присущи те же плюсы, что и веб-приложениям вообще, не связанным с машинным обучением.

- В отличие от нативных приложений созданные с помощью TensorFlow.js JavaScript-приложения будут работать на множестве разных семейств устройств: от стационарных компьютеров Mac, Windows и Linux до устройств на Android и iOS.
- Благодаря оптимизации 2D- и 3D-графики браузер — наиболее функциональная и развитая среда для интерактивной визуализации данных. Где бы ни нужно было продемонстрировать поведение и внутреннее устройство нейронных сетей людям, сложно представить себе среду лучше браузера. Рассмотрим, например, Playground («песочницу») TensorFlow (https://playground.tensorflow.org/). Это чрезвычайно популярное веб-приложение, в котором можно интерактивно решать задачи классификации с помощью нейронных сетей. Можно модифицировать структуру и гиперпараметры нейронной сети, наблюдая, как при этом меняются ее скрытые слои и выходные сигналы (рис. 1.6). Если вы еще не экспериментировали с ней — попробуйте обязательно. По мнению многих, это один из наиболее наглядных и восхитительных учебных инструментов по тематике нейронных сетей. Playground TensorFlow, по сути, является предшественником TensorFlow.js. Хотя и потомок Playground — библиотека TensorFlow.js — обладает намного более широкими возможностями глубокого обучения и намного лучшей оптимизацией быстродействия. Кроме того, она оснащена специализированным компонентом для визуализации моделей глубокого обучения (мы рассмотрим его подробно в главе 7). TensorFlow.js очень пригодится вам, неважно, требуется

создать простые образовательные приложения наподобие Playground TensorFlow или продемонстрировать результаты исследования на тему глубокого обучения в наиболее наглядном и интуитивно понятном виде (см. примеры, в частности визуализацию вложений t-SNE в режиме реального времени[1]).

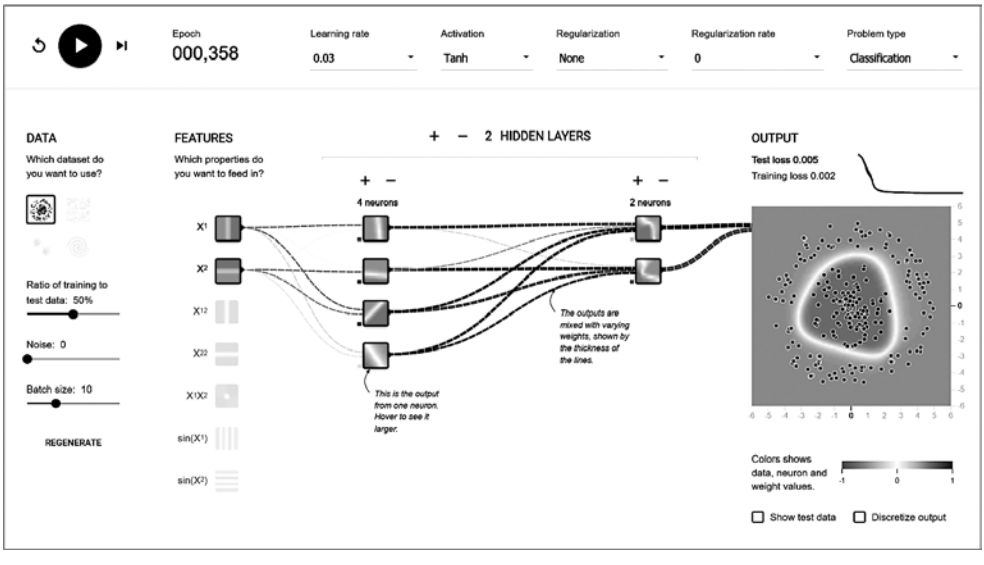


Рис. 1.6. Снимок экрана Playground TensorFlow (https://playground.tensorflow.org/) — популярного браузерного UI для обучения работе нейронных сетей от Дэниела Смилкова и его коллег из Google. Playground TensorFlow — один из главных предшественников проекта TensorFlow.js

1.2.1. Глубокое обучение с помощью Node.js

Из соображений безопасности и быстродействия браузеры представляют собой среды с ограничением ресурсов, в частности объема доступной им оперативной и дисковой памяти. Это значит, что браузер не идеальная среда для обучения больших моделей ML со значительными объемами данных, хотя он и идеален для множества видов задач вывода, обучения в небольших масштабах и переноса обучения, требующих меньших ресурсов. Впрочем, Node.js полностью меняет ситуацию. Благодаря Node.js код на JavaScript можно выполнять вне браузера, получая таким образом доступ ко всем нативным ресурсам, в частности к RAM и файловой системе. В TensorFlow.js включена одна из версий Node.js, а именно *tfjs-node*. Она привязывается непосредственно к нативным библиотекам TensorFlow, скомпилированным из кода C++ и CUDA. Это дает возможность пользователям работать с теми же самыми распараллеленными ядрами CPU и GPU, что и внутри TensorFlow (на языке Python). Можно показать эмпирически, что скорость обучения модели в tfjs-node

[1] См.: *Pezzotti N.* Realtime tSNE Visualizations with TensorFlow.js // googblogs. http://mng.bz/nvDg.

сравнима со скоростью Keras в Python. А значит, tfjs-node — подходящая среда для обучения больших моделей ML со значительными объемами данных. В книге приводятся примеры, в которых мы обучаем с помощью tfjs-node такие модели, которые не по зубам браузеру (например, модель для распознавания слов в главе 5 и средство анализа тональности текста в главе 9).

Но почему вообще имеет смысл предпочесть Node.js более широко известной среде Python для обучения моделей машинного обучения? Ответов два: 1) быстродействие; 2) совместимость с уже существующим стеком и набором профессиональных навыков программистов. Во-первых, что касается быстродействия, то новейшие интерпретаторы JavaScript (например, используемый Node.js движок V8) выполняют динамическую (JIT) компиляцию кода на JavaScript, что обеспечивает более высокую, по сравнению с Python, производительность. В результате обучение модели в tfjs-node зачастую занимает меньше времени, чем в Keras (Python), если эта модель достаточно мала для того, чтобы определяющим фактором была производительность интерпретатора языка.

Во-вторых, Node.js — очень популярная среда для создания серверных приложений. Если ваша серверная часть уже написана на Node.js и вы хотите добавить в свой стек машинное обучение, будет разумнее воспользоваться tfjs-node, а не Python. Благодаря тому что весь код написан на одном языке, можно напрямую переиспользовать большие фрагменты базы кода, включая функции загрузки и форматирования данных, а значит, ускорить подготовку конвейера обучения модели. А поскольку в стек не добавляется новый язык, его сложность и затраты на сопровождение снижаются и, возможно, не придется дополнительно нанимать программиста на Python.

Наконец, написанный на TensorFlow.js код машинного обучения будет работать как в среде браузера, так и в Node.js, за исключением, возможно, кода обработки данных, использующего API, работающие исключительно в браузере или Node. Почти все примеры кода из книги будут работать в обеих средах. Мы приложили все усилия, чтобы отделить не зависящий от среды, ориентированный на машинное обучение код от ориентированного на конкретную среду кода ввода и обработки данных. Дополнительное преимущество состоит в том, что достаточно изучить одну библиотеку, чтобы проводить глубокое обучение как на стороне сервера, так и на стороне клиента.

1.2.2. Экосистема JavaScript

При оценке того, подходит ли JavaScript для определенного типа приложений, например для глубокого обучения, не стоит игнорировать исключительно обширную экосистему этого языка. На протяжении многих лет JavaScript неизменно занимал первое место среди нескольких десятков языков программирования по количеству репозиториев и запросов на включение кода на GitHub (см. http://githut.info/). В системе управления пакетами npm, де-факто главном открытом репозитории пакетов JavaScript, по состоянию на июль 2018 года насчитывалось более 600 000 пакетов. Это более чем вчетверо превышает количество пакетов в PyPI, фактически главном открытом репозитории пакетов Python (http://www.modulecounts.com/). И хотя у ма-

шинного обучения и науки о данных на Python и R более сложившееся сообщество, сообщество разработчиков на JavaScript тоже постепенно наращивает поддержку конвейеров обработки данных для машинного обучения.

Хотите вводить и обрабатывать данные из облачных хранилищ и баз данных? И Google Cloud, и Amazon Web Services предоставляют API для Node.js. Большинство наиболее популярных сегодня систем баз данных, например MongoDB и RethinkDB, полноценно поддерживают драйверы Node.js. Хотите выполнить первичную обработку данных на JavaScript? Мы рекомендуем книгу *Data Wrangling with JavaScript* Эшли Дэвиса (Davis Ashley. Manning Publications, 2018, https://www.manning.com/books/data-wrangling-with-javascript). Хотите визуализировать данные? Существуют такие отработанные и эффективные библиотеки, как d3.js, vega.js и plotly.js, во многом превосходящие библиотеки визуализации языка Python. Достаточно подготовить входные данные — и TensorFlow.js, которой посвящена эта книга, возьмет на себя все остальное и поможет вам создать, обучить и реализовать модели глубокого обучения, а также сохранить, загрузить и визуализировать их.

Наконец, экосистема JavaScript постоянно развивается самым перспективным образом. Она теперь охватывает не только свои традиционные оплоты — а именно, браузер и прикладную среду Node.js, — но и новые для себя территории, например традиционные, не веб-, приложения (Electron) и нативные мобильные приложения (React Native и Ionic). Зачастую писать UI и приложения для подобных фреймворков проще, чем использовать бесчисленные утилиты создания приложений для конкретных платформ. JavaScript — язык, у которого есть потенциал для распространения возможностей глубокого обучения на все платформы. Мы вкратце резюмируем основные преимущества сочетания JavaScript с глубоким обучением в табл. 1.2.

Таблица 1.2. Краткое резюме преимуществ выполнения глубокого обучения на JavaScript

Соображения	Примеры
Причины, относящиеся к клиентской стороне	Снижение задержки вывода и обучения за счет локальности данных. Возможность работы моделей даже при отключении клиента. Защита персональной информации (данные не покидают браузер). Уменьшение затрат на серверы. Упрощенный стек развертывания
Причины, относящиеся к браузеру	Доступность разнообразных типов входных данных (видео- и аудио-API HTML5, а также API датчиков) для вывода и обучения. Не нужно устанавливать. Доступность параллельных вычислений без необходимости установки благодаря API WebGL для широкого диапазона GPU. Поддержка на множестве платформ. Идеальная интерактивная среда для визуализации. Пронизанная естественными взаимосвязями среда открывает непосредственный доступ ко множеству разнообразных источников данных и ресурсов для машинного обучения

Продолжение ⇩

Таблица 1.2 *(продолжение)*

Соображения	Примеры
Причины, связанные с JavaScript	JavaScript — по многим оценкам, наиболее популярный язык программирования с открытым исходным кодом, благодаря чему есть много талантливых энтузиастов, предпочитающих этот язык. JavaScript отличается процветающей экосистемой и множеством приложений как на клиентской, так и на серверной стороне. Node.js дает возможность выполнять приложения на серверной стороне без присущих браузерам ограничений ресурсов. Движок V8 обеспечивает быструю работу JavaScript-кода

1.3. Почему именно TensorFlow.js

Чтобы проводить глубокое обучение на JavaScript, необходимо сначала выбрать библиотеку. Для книги мы выбрали библиотеку TensorFlow.js. В этом разделе мы расскажем, что представляет собой TensorFlow.js и почему мы выбрали именно ее.

1.3.1. Краткая история TensorFlow, Keras и TensorFlow.js

TensorFlow.js — библиотека, с помощью которой можно проводить глубокое обучение на языке JavaScript. Как понятно из названия, TensorFlow.js спроектирована в расчете на полную совместимость с TensorFlow — фреймворком языка Python для глубокого обучения. Чтобы разобраться с TensorFlow.js, нам придется вкратце описать историю TensorFlow.

TensorFlow был создан как библиотека с открытым исходным кодом в ноябре 2015 года командой специалистов по глубокому обучению из компании Google. В числе участников были и авторы данной книги. С момента появления популярность TensorFlow сильно выросла. Сейчас он применяется для широкого спектра промышленных приложений и исследовательских проектов как в Google, так и за его стенами, более широким сообществом разработчиков. Название *TensorFlow*[1] призвано отражать происходящее внутри типичной программы, написанной на его основе: представления данных, так называемые *тензоры*, проходят через слои и другие узлы обработки данных, за счет чего выполняется обучение моделей ML и вывод на их основе.

Прежде всего, что такое тензор? Это просто «многомерный массив» в терминологии специалистов по вычислительной технике. В нейронных сетях и глубоком обучении все данные и результаты вычислений представляются в виде тензоров. Например, изображение в оттенках серого можно представить в виде двумерного массива чисел — двумерного тензора, цветное изображение обычно представляется в виде трехмерного тензора, дополнительное измерение в котором используется для цветовых каналов. Звук, видео, текст и все прочие виды данных можно представить в виде тензоров. У любого тензора есть два основных свойства: тип данных (напри-

[1] Образовано от слов «тензор» и «поток данных». — *Примеч. пер.*

мер, `float32` или `int32`) и форма. Форма определяет размер тензора по каждому измерению. Например, форма двумерного тензора может быть `[128, 256]`, а трехмерного — `[10, 20, 128]`. Данные, превращенные в тензор с конкретным типом и формой, можно подать на вход любого слоя, принимающего данные этого типа и формы, вне зависимости от их исходного смысла. Таким образом, тензоры играют роль общего языка, понятного всем моделям глубокого обучения.

Но почему именно тензоры? Из предыдущего раздела мы узнали, что основной объем вычислений, необходимых для работы глубокой нейронной сети, производится в виде массово-параллельных операций, обычно на GPU, что означает выполнение одной и той же вычислительной операции над множеством элементов данных. Тензоры играют роль контейнеров для организации данных в структуры, удобные для параллельной обработки. Совершенно очевидно, что при сложении тензора А формы `[128, 128]` с тензором Б формы `[128, 128]` необходимо выполнить 128×128 независимых друг от друга операций сложения.

А что же насчет *flow*? Представьте себе, что тензор — жидкость, переносящая данные. В TensorFlow данные «текут» через *граф* — структуру данных, состоящую из связанных между собой математических операций *(узлов)*. Как демонстрирует рис. 1.7, узел может представлять собой последовательные слои нейронной сети. Все узлы получают на входе тензоры и возвращают тензоры. «Тензорная жидкость» преобразуется в различные формы и значения по мере того, как «течет» по графу TensorFlow. Это соответствует преобразованию представлений, то есть основной задаче нейронных сетей, как мы упоминали в предыдущих разделах. С помощью TensorFlow специалисты по машинному обучению могут создавать любые виды нейронных сетей: от неглубоких до очень глубоких, от сверточных нейронных сетей для машинного зрения до рекуррентных нейронных сетей (RNN) для задач преобразования последовательностей в последовательности. Графовые структуры данных можно сериализовать и развертывать на множестве различных типов устройств, от мейнфреймов до мобильных телефонов.

TensorFlow по своей сути сделан очень универсальным и гибким: операции могут быть любыми четко определенными математическими функциями, а не только слоями нейронных сетей. В частности, они могут быть низкоуровневыми математическими операциями наподобие сложения и умножения тензоров — как раз теми операциями, которые происходят *внутри* слоя нейронной сети. Благодаря этому специалисты по глубокому обучению и исследователи могут свободно описывать произвольные новые операции для глубокого обучения. Впрочем, для многих занимающихся глубоким обучением подобные низкоуровневые манипуляции не стоят свеч. Они приводят к тому, что код становится раздутым и подверженным ошибкам, а также удлиняют цикл разработки. Большинство специалистов по глубокому обучению применяют лишь несколько конкретных типов слоев (например, свертку, субдискретизацию или плотные слои, как вы увидите в следующих главах). Создавать новые типы слоев им приходится очень редко. Здесь уместна аналогия с LEGO. В конструкторах LEGO насчитывается всего несколько видов кирпичиков. Собирающему конструктор не нужно думать, как сделать кирпичик LEGO. Чего нельзя сказать о таких игрушках, как, скажем, пластилин Play-Doh, которые можно сравнить с низкоуровневым API TensorFlow. Тем не менее существует комбинаторно большое число способов соединения кирпичиков, а значит, и практически

безграничные возможности для творчества. Игрушечный дом можно построить как из кирпичиков LEGO, так и из пластилина Play-Doh, но если у вас нет каких-то очень специфических требований к размеру, форме, текстуре или материалу дома, гораздо проще и быстрее построить его с помощью LEGO. В большинстве случаев дома из кирпичиков LEGO окажутся устойчивее и изящнее, чем из Play-Doh.

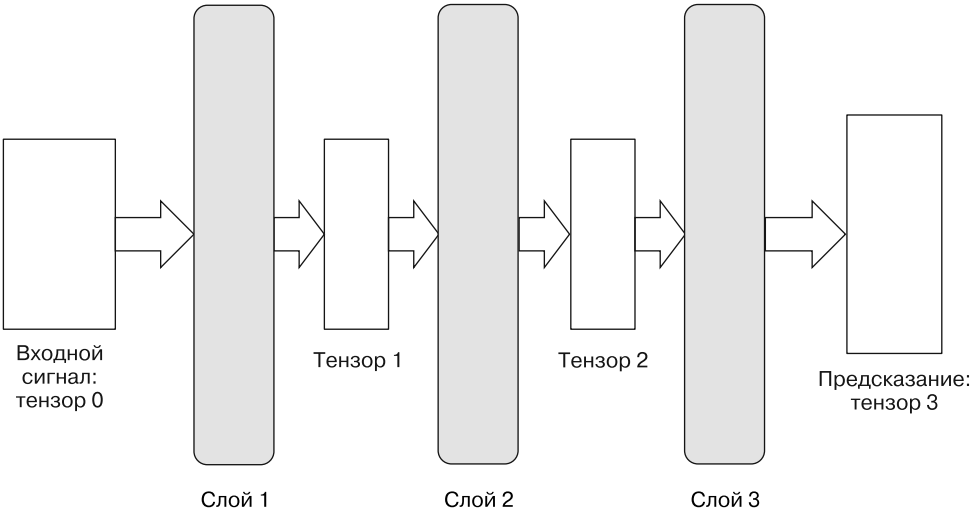


Рис. 1.7. Тензоры «текут» через несколько слоев — распространенный сценарий в TensorFlow и TensorFlow.js

В мире TensorFlow эквивалентом LEGO является высокоуровневый API — Keras[1]. Keras предоставляет набор наиболее распространенных типов слоев нейронных сетей, содержит настраиваемые параметры. Он дает возможность связывать слои в нейронные сети. Кроме того, Keras включает API для того, чтобы:

- задавать нюансы обучения нейронной сети (функций потерь, метрик и оптимизаторов);
- подавать данные для обучения или оценки работы нейронной сети, а также использовать модель для вывода;
- отслеживать процесс обучения (с помощью обратных вызовов);
- сохранять и загружать модели;
- выводить в текстовом или графическом виде архитектуру моделей.

[1] На самом деле с момента возникновения TensorFlow появилось несколько высокоуровневых API, созданных как специалистами из Google, так и сообществом разработчиков открытого исходного кода. Наиболее популярные из них — Keras, tf.Estimator, tf.contrib.slim и TensorLayers. Для читателей данной книги важнейшим из высокоуровневых API TensorFlow, конечно, является Keras, поскольку высокоуровневый API TensorFlow.js создавался по образу и подобию Keras и TensorFlow.js обеспечивает двустороннюю совместимость с Keras в смысле сохранения/загрузки моделей.

При использовании Keras для реализации полного технологического процесса глубокого обучения хватает нескольких строк кода. Благодаря гибкости низкоуровневых и удобству использования высокоуровневых API TensorFlow и Keras образуют ведущую (в смысле использования в промышленности и научной среде) экосистему глубокого обучения (см. твит http://mng.bz/vlDJ). Не стоит недооценивать их роль в обеспечении доступности глубокого обучения для широкой публики. До появления фреймворков наподобие TensorFlow и Keras осуществлять глубокое обучение на практике могли только специалисты с навыками программирования CUDA и обширным опытом написания нейронных сетей на C++. Благодаря TensorFlow и Keras создание глубоких нейронных сетей с GPU-ускорением требует намного меньше навыков и усилий. Но оставалась одна проблема: раньше было невозможно реализовывать модели TensorFlow или Keras непосредственно в браузере. Для выдачи обученных моделей DL в браузере приходилось выполнять HTTP-запросы к серверу прикладной части. Именно здесь и пригодилась TensorFlow.js. Создателями проекта TensorFlow.js были Нихиль Торат и Дэниел Смилков — два эксперта по визуализации данных глубокого обучения и человеко-машинному взаимодействию[1] из компании Google. Как мы уже упоминали, начало проекта TensorFlow.js заложила чрезвычайно популярная демонстрационная модель глубоких нейронных сетей TensorFlow Playground. В сентябре 2017 года была выпущена библиотека deeplearn.js с API, аналогичным низкоуровневому API TensorFlow. Она поддерживала операции над нейронными сетями с WebGL-ускорением, обеспечивая работу реальных нейронных сетей в браузере с низкими задержками вывода.

После первых успехов deeplearn.js к проекту присоединились и другие участники команды Google Brain, и он был переименован в TensorFlow.js. JavaScript API подвергся серьезной переработке для усиления его совместимости с TensorFlow. Кроме того, поверх низкоуровневого ядра был надстроен высокоуровневый API а-ля Keras, что значительно упростило для пользователей описание, обучение и выполнение моделей глубокого обучения в этой JavaScript-библиотеке. Сегодня все сказанное выше относительно мощи и удобства использования Keras справедливо и для TensorFlow.js. Для дальнейшего расширения совместимости были созданы средства преобразования для импорта в TensorFlow.js моделей, сохраненных из TensorFlow и Keras, а равно и экспорта их оттуда. После первой его демонстрации на конференциях TensorFlow Developer Summit и Google I/O весной 2018-го (см. www.youtube.com/watch?v=YB-kfeNIPCE и www.youtube.com/watch?v=OmofOvMApTU), TensorFlow.js быстро стала чрезвычайно популярной библиотекой глубокого обучения на JavaScript с наибольшим на сегодня количеством веток среди всех подобных библиотек на GitHub.

На рис. 1.8 приведена общая архитектура TensorFlow.js. Нижний уровень отвечает за быстрые параллельные вычисления математических операций. Хотя большинству пользователей этот уровень не виден, его высокая производительность чрезвычайно важна и позволяет обеспечить максимально быстрое обучение модели и вывод на более высоких уровнях API. В браузере для GPU-ускорения используется WebGL (см. инфобокс 1.2). В Node.js доступны как прямые привязки для распараллеливания с помощью многоядерного процессора, так и GPU-ускорение

[1] Интересная историческая справка: эти авторы также сыграли ключевую роль в создании TensorBoard — популярного инструмента визуализации моделей TensorFlow.

на основе CUDA. Это те же самые математические прикладные части, что используются TensorFlow и Keras в языке Python. На основе самого нижнего математического уровня строится *API Ops*, вполне соответствующий низкоуровневому API TensorFlow и поддерживающий загрузку моделей в формате SavedModels из TensorFlow. На верхнем уровне располагается *API Layers* в стиле Keras. Он отлично подходит для большинства программистов, работающих с TensorFlow.js. API Layers также поддерживает импорт/экспорт моделей Keras.

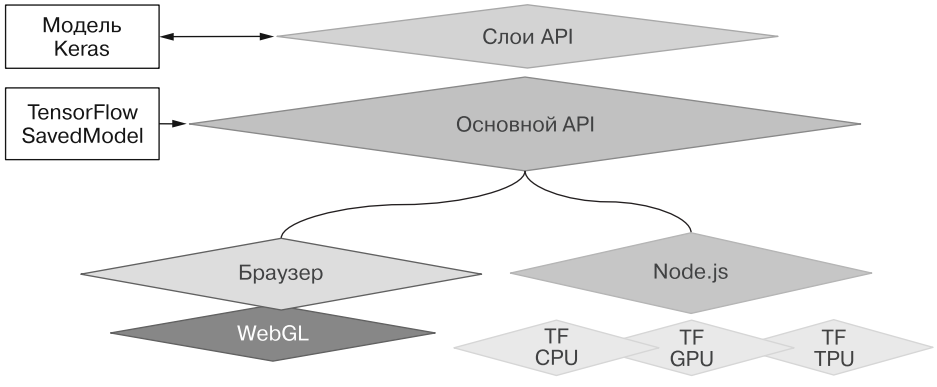


Рис. 1.8. Общий обзор архитектуры TensorFlow.js. Показаны также ее взаимосвязи с библиотеками TensorFlow и Keras языка Python

1.3.2. Почему именно TensorFlow.js: краткое сравнение с аналогичными библиотеками

TensorFlow.js не единственная JavaScript-библиотека для глубокого обучения и даже не первая (например, история библиотек brain.js и ConvNetJS намного продолжительнее). Так почему же TensorFlow.js выделяется среди себе подобных? Первая причина состоит в ее всесторонности: TensorFlow.js — единственная доступная в настоящий момент библиотека, которая поддерживает все ключевые составляющие реального технологического процесса глубокого обучения:

- обеспечивает как вывод, так и обучение;
- поддерживает браузеры и Node.js;
- использует GPU-ускорение (WebGL в браузерах и ядра CUDA в Node.js);
- поддерживает описание архитектур моделей нейронных сетей на JavaScript;
- обеспечивает сериализацию и десериализацию моделей;
- поддерживает преобразование моделей из фреймворков глубокого обучения на Python и в них;
- снабжена встроенной поддержкой ввода/обработки данных и API визуализации.

Вторая причина — экосистема. В большинстве библиотек глубокого обучения JavaScript описан свой собственный уникальный API, в то время как TensorFlow.js тес-

но интегрирована с TensorFlow и Keras. У вас есть обученная модель из TensorFlow или Keras на Python и вы хотели бы использовать ее в браузере? Никаких проблем. Создали модель TensorFlow.js в браузере и хотите перенести ее в Keras для доступа к более быстрым аппаратным средствам вычислений, например TPU от Google? И это возможно! Тесная интеграция с фреймворками других языков, не JavaScript, не только повышает совместимость, но и упрощает разработчикам миграцию между мирами разных языков программирования и стеков инфраструктуры. Например, когда вы изучите TensorFlow.js по этой книге, то с легкостью сможете начать использовать Keras на Python. Обратный путь также не представляет сложностей: любой разработчик со знанием Keras сможет быстро изучить TensorFlow.js (при наличии достаточных навыков работы с JavaScript). И последнее по порядку, но отнюдь не по значимости: не стоит упускать из виду популярность TensorFlow.js и многочисленность его сообщества. Разработчики TensorFlow.js всецело преданы идеям ее долгосрочного сопровождения и поддержки. С TensorFlow.js не сравнится ни одна из конкурирующих библиотек ни по количеству веток на GitHub, ни по количеству внешних участников проекта, ни по живости обсуждений на форумах, ни по числу вопросов и ответов на Stack Overflow.

1.3.3. Как TensorFlow.js используется в мире

Лучшее свидетельство мощи и популярности библиотеки — ее применение в реальных приложениях. Вот несколько заслуживающих упоминания примеров использования TensorFlow.js.

- В проекте Magenta компании Google с помощью TensorFlow.js RNN и другие глубокие нейронные сети генерируют музыку к фильмам и новые звуки музыкальных инструментов (см. https://magenta.tensorflow.org/demos).
- Дэн Шиффман и его соратники из Нью-Йоркского университета создали ML5.js — удобный в использовании высокоуровневый API для различных готовых моделей глубокого обучения, работающих в браузере, например, для обнаружения объектов и переноса стиля изображений (https://ml5js.org/).
- Абхишек Сингх — разработчик программного обеспечения с открытым исходным кодом — создал браузерный интерфейс для перевода американского жестового языка в речь, чтобы помочь немым или глухим людям пользоваться «умными» колонками наподобие Amazon Echo[1].
- Основанное на TensorFlow.js веб-приложение Canvas Friends помогает учиться рисовать (www.y8.com/games/canvas_friends).
- MetaCar, браузерный симулятор беспилотного автомобиля, использует TensorFlow.js для реализации жизненно необходимых для его симуляций алгоритмов обучения с подкреплением (www.metacar-project.com).
- Приложение Doctor из набора утилит Clinic.js, основанное на Node.js приложение для мониторинга производительности серверных программ, реализует

[1] *Singh A.* Getting Alexa to Respond to Sign Language Using Your Webcam and TensorFlow.js // Medium, 8 Aug. 2018. http://mng.bz/4eEa.

с помощью TensorFlow.js скрытую модель Маркова и использует ее для выявления всплесков использования CPU[1].

- Обратите также внимание на прочие созданные сообществом разработчиков открытого исходного кода потрясающие приложения по адресу https://github.com/tensorflow/tfjs/blob/master/GALLERY.md.

1.3.4. Что вы узнаете о TensorFlow.js из этой книги, а что — нет

Прочитав книгу, вы научитесь создавать с помощью TensorFlow.js приложения наподобие следующих.

- Веб-сайт, классифицирующий загружаемые пользователем изображения.
- Глубокие нейронные сети, получающие визуальные и аудиоданные от связанных с браузером датчиков и выполняющие над ними различные задачи машинного обучения в режиме реального времени, например распознавание и перенос обучения.
- ИИ для работы с естественным языком на стороне клиента, например классификатор тональностей комментариев для возможности их модерации.
- Программа обучения модели ML Node.js, использующая данные в гигабайтных объемах и GPU-ускорение.
- Обучаемый с подкреплением алгоритм на основе TensorFlow.js для решения мелких задач, связанных с управлением и играми.
- Инструментальная панель для демонстрации внутреннего устройства обученных моделей и результатов экспериментов с ними при машинном обучении.

Что особенно важно — вы не только научитесь создавать подобные приложения, но и начнете понимать, как они работают. Например, вы узнаете суть стратегий и ограничений создания моделей глубокого обучения для различных видов задач. Кроме того, прочитаете об этапах и нюансах обучения и развертывания подобных моделей.

Машинное обучение — обширная сфера знаний, а TensorFlow.js — универсальная библиотека. Поэтому с помощью существующих в TensorFlow.js технологий вполне можно создать приложения, которые выходят за рамки данной книги. В их числе:

- высокопроизводительное, распределенное обучение глубоких нейронных сетей на колоссальных объемах данных (порядка терабайтов) в среде Node.js;
- методики, не связанные с нейронными сетями, например SVM, деревья принятия решений и случайные леса;
- продвинутые приложения глубокого обучения, например системы автоматического реферирования текста, позволяющие резюмировать большой документ в нескольких репрезентативных предложениях, системы преобразования изображений в текст, генерирующие текстовое описание входных изображений,

[1] *Madsen A.* Clinic.js Doctor Just Got More Advanced with TensorFlow.js // Clinic.js blog, 22 Aug. 2018. http://mng.bz/Q06w.

а также генеративные модели для изображений, увеличивающие разрешение входных изображений.

Впрочем, эта книга даст вам базовые знания глубокого обучения, с которыми вы сможете сами изучить статьи и код, связанные с вышеупомянутыми продвинутыми приложениями.

Как и у любой другой технологии, у TensorFlow.js есть свои ограничения. Некоторые задачи ей не под силу. И хотя в будущем эти границы, вероятно, раздвинутся, не помешает четко себе представлять их по состоянию на текущий момент.

- Выполнение моделей глубокого обучения, требования которых к памяти превышают ограничения RAM и WebGL для отдельной вкладки браузера. При выполнении вывода в браузере это обычно означает модель с суммарным объемом весов более ~100 Мбайт. Для обучения требуется больше памяти и вычислительных ресурсов, так что вполне вероятно, что обучение даже меньших моделей на вкладке браузера будет происходить слишком медленно. Обучение модели обычно требует бóльших объемов данных, чем вывод, — еще один ограничивающий фактор, который необходимо учитывать при оценке осуществимости обучения в браузере.
- Создание высокопроизводительных моделей обучения с подкреплением, например способных обыграть человека в го.
- Распределенное (на нескольких машинах) обучение моделей DL с помощью Node.js.

Упражнения

1. Неважно, разработчик вы клиентской части на JavaScript или Node.js, попробуйте обсудить несколько возможных сценариев применения машинного обучения к системе, над которой вы сейчас работаете, чтобы сделать ее более «интеллектуальной». Почерпнуть идеи вы можете из табл. 1.1 и 1.2, а также из подраздела 1.3.3. Вот еще несколько примеров.
   А. Веб-сайт для продажи аксессуаров наподобие солнечных очков, где изображения лиц пользователей захватываются с помощью веб-камеры и выявляются ключевые точки лиц с помощью глубокой нейронной сети, работающей на основе TensorFlow.js. Выявленные ключевые точки затем используются для синтеза изображения с наложенными на лицо пользователя солнечными очками для имитации на веб-странице примерки очков. Эта примерка достаточно реалистична, поскольку благодаря выводу на стороне клиента она происходит с низким временем задержки и высокой частотой кадров. А поскольку захваченный снимок лица не покидает браузера, гарантируется защита персональной информации пользователя.
   Б. Мобильное спортивное приложение, написанное на React Native (кроссплатформенная JavaScript-библиотека для создания нативных мобильных приложений), для мониторинга спортивных тренировок пользователей. С помощью API HTML5 это приложение получает данные в режиме реального времени с гироскопа и акселерометра телефона. Эти данные пропускаются

через основанную на TensorFlow.js модель, автоматически определяющую вид тренировок пользователя (например, отдых или ходьба, бег трусцой или спринт).

В. Расширение для браузера, автоматически определяющее, кто использует устройство — взрослый или ребенок (по захваченным с веб-камеры с частотой 1 кадр в 5 секунд изображениям с помощью модели машинного зрения на основе TensorFlow.js), и предоставляющее/блокирующее на основе этой информации доступ к определенным сайтам.

Г. Браузерная среда программирования, в которой рекуррентная нейронная сеть, реализованная с помощью TensorFlow.js, находит опечатки в комментариях к коду.

Д. Серверное приложение на основе Node.js для координации грузового логистического сервиса, в котором прогнозируется ожидаемое время прибытия (ETA) для каждой транзакции на основе поступающих в реальном времени сигналов, в частности, о состоянии перевозчика, типе и количестве груза, дате/времени, а также информации о дорожном трафике. Конвейеры обучения и вывода написаны на Node.js с помощью TensorFlow.js, что упрощает серверный стек.

Резюме

- Сфера ИИ охватывает исследования автоматизации когнитивных задач. Машинное обучение — подобласть ИИ, в которой правила решения задач наподобие классификации изображений подбираются автоматически, путем усвоения моделью примеров из обучающих данных.
- Основная задача машинного обучения — преобразование исходного представления данных в представление, лучше подходящее для решения конкретной задачи.
- Нейронные сети — подход из сферы машинного обучения, в котором преобразование представления данных выполняется за счет последовательных шагов (слоев) математических операций. Область глубокого обучения охватывает глубокие нейронные сети — нейронные сети с большим количеством слоев.
- Благодаря развитию аппаратного обеспечения, большей доступности маркированных данных и усовершенствованию алгоритмов область глубокого обучения продемонстрировала колоссальный прогресс с начала 2010-х годов, позволив решить ранее неразрешимые задачи и создав потрясающие новые перспективы.
- Браузер — удобная среда для развертывания и обучения глубоких нейронных сетей на базе JavaScript.
- TensorFlow.js, которой посвящена эта книга, — универсальная, всесторонняя и обладающая очень широкими возможностями библиотека с открытым исходным кодом для глубокого обучения на JavaScript.

Часть II

Введение в TensorFlow.js

Мы уже рассмотрели основы TensorFlow.js и в этой части книги приступим к изучению машинного обучения на практике. В главе 2 начнем с простой задачи машинного обучения — регрессии (предсказания одного числа) — и постепенно перейдем к более сложным задачам, например бинарной и многоклассовой классификации (см. главы 3 и 4). В полном соответствии с типами решаемых задач мы плавно перейдем от простых данных («плоских» массивов чисел) к более сложным (изображениям и аудиоданным). По мере изучения конкретных задач и кода для их решения обсудим математические основы таких методов, как обратное распространение ошибки.

В книге мы отказались от формальных математических описаний и стараемся пояснять все понятия с помощью графиков и псевдокода. В главе 5 мы обсудим перенос обучения — эффективный способ переиспользования предобученных нейронных сетей для работы с новыми данными, а также разберем подход, специально предназначенный для браузерной среды глубокого обучения.

2 Приступим: простая линейная регрессия в TensorFlow.js

В этой главе

- Пример нейронной сети для простой задачи машинного обучения — линейной регрессии.
- Тензоры и операции над ними.
- Основные методы оптимизации нейронных сетей.

Никто не любит ждать, особенно если неизвестно, сколько именно придется это делать. Любой специалист, занимающийся UX-дизайном (User Experience — взаимодействие с пользователем), подтвердит, что если избавиться от задержки невозможно, то лучшее, что можно сделать, — предоставить пользователям информацию о времени ожидания. Оценка ожидаемых задержек — задача предсказания, и с помощью библиотеки TensorFlow.js можно сделать точную оценку времени скачивания с учетом контекста. Так пользователи будут чувствовать уважение к их времени и внимание со стороны разработчиков системы.

В этой главе на простом примере предсказания времени скачивания мы продемонстрируем основные компоненты законченной модели машинного обучения. В рассказе мы с практической точки зрения охватим тензоры, моделирование и оптимизацию, чтобы вы поняли, что они собой представляют, как работают и как их правильно использовать.

Чтобы разобраться в глубоком обучении на уровне исследователя, посвятившего этому годы жизни, необходимо овладеть множеством математических дисциплин.

Впрочем, для специалиста-практика знакомство с линейной алгеброй, дифференциальным исчислением и статистикой многомерных пространств желательно, но не обязательно, даже при создании сложных, высокопроизводительных систем. Наша цель в этой главе и вообще в книге — познакомить вас с различными техническими моментами, где это возможно, с помощью кода, а не математических формул. Мы хотим научить вас понимать принципы работы на интуитивном уровне, без специальных познаний предметной области.

2.1. Пример 1. Предсказание продолжительности скачивания с помощью TensorFlow.js

Что ж, приступим! Создадим простейшую нейронную сеть на основе библиотеки TensorFlow.js (это название иногда сокращают до tfjs) для предсказания времени скачивания по размеру загружаемого файла. Если вы еще не сталкивались с TensorFlow.js или ей подобными библиотеками, то вряд ли сразу все поймете в примере, и это нормально. Все приведенные понятия мы подробно обсудим в последующих главах, так что не волнуйтесь, если что-то покажется вам странным. С чего-то ведь нужно начинать. И мы начнем с написания короткой программы, принимающей в качестве входных данных размер файла и выводящей прогнозируемое время скачивания этого файла.

2.1.1. Обзор проекта: предсказание продолжительности

При первом близком знакомстве с системой машинного обучения вас может напугать разнообразие новых понятий и терминов. Следовательно, имеет смысл сначала взглянуть на технологический процесс в целом. Схематически данный пример описан на рис. 2.1. Это паттерн, который будет использоваться для всех примеров книги.

Во-первых, мы получим доступ к нашим обучающим данным. При машинном обучении данные могут читаться с диска, скачиваться по сети, генерироваться или просто «зашиваться» в программу. В этом примере мы воспользуемся последним из перечисленных подходов, поскольку он удобен, а нам нужно лишь небольшое количество данных. Первый шаг — создание модели, которое, как мы видели в главе 1, напоминает проектирование подходящей обучаемой функции: функции, отображающей входные данные в предсказываемые величины. В данном случае входные и прогнозируемые данные представляют собой числа. Когда у нас будут и модель, и данные, мы обучим модель, отслеживая по ходу дела ее метрики. И наконец, воспользуемся обученной моделью для предсказания на основе еще не встречавшихся ей данных и оценим показатель ее безошибочности.

Мы пройдем по всем этим этапам, приводя работоспособные фрагменты кода, которые вы сможете просто скопировать, с пояснениями теории и конкретных инструментов.

Рис. 2.1. Обзор основных этапов системы предсказания времени скачивания — нашего первого примера

2.1.2. Примечания относительно листингов и команд консоли

Код в этой книге приводится в двух форматах. *Листинги* представляют собой структурированный код, находящийся в указанных в тексте репозиториях. У каждого листинга свое название и номер. Например, листинг 2.1 содержит очень короткий фрагмент HTML-кода, который вы можете скопировать в неизменном виде в файл,

например /tmp/tmp.html, на своем компьютере и затем открыть в своем браузере, перейдя по адресу file:///tmp/tmp.html, хотя сам по себе он мало что делает.

Команды консоли (console interaction) — это менее формальные блоки кода, воспроизводящие примеры диалоговых действий в JavaScript REPL[1], например в консоли JavaScript браузера (открывается нажатием Cmd-Opt-J, Ctrl+Shift+J или F12 в Chrome, впрочем, в вашем браузере/ОС клавиши быстрого запуска могут быть другими). Командам консоли предшествует знак >, как в Chrome или Firefox, а результаты их работы приводятся в следующей строке. Например, следующие команды консоли создают массив и выводят его значение. Результаты, которые будут выведены в вашей консоли JavaScript, могут немного отличаться, но суть останется той же:

```
> let a = ['hello', 'world', 2 * 1009]
> a;
(3) ["hello", "world", 2018]
```

Лучший способ проверить, выполнить и изучить листинги кода из книги — клонировать указанные репозитории и затем поэкспериментировать с ними. При написании книги мы регулярно работали с CodePen — простым, интерактивным, допускающим совместное использование репозиторием (http://codepen.io/). Например, листинг 2.1 доступен для ваших экспериментов по адресу codepen.io/tfjs-book/pen/VEVMbx. При переходе на CodePen он должен выполниться автоматически, а вы — увидеть вывод в консоли. Щелкните на вкладке Console снизу слева, чтобы открыть консоль. Если код в CodePen не запускается автоматически, попробуйте внести маленькое, несущественное изменение, например добавьте пробел в конце, — это подтолкнет репозиторий к работе.

Листинги из этого раздела доступны в следующей коллекции CodePen: codepen.io/collection/Xzwavm/. CodePen отлично справляется, когда речь идет об отдельном файле JavaScript, но более объемные примеры с более широкой структурой хранятся в репозиториях GitHub, как вы увидите далее. Что касается этого примера, мы рекомендуем прочитать весь раздел, а затем экспериментировать с соответствующими CodePen.

2.1.3. Создание и форматирование данных

Давайте оценим, сколько времени займет скачивание файла на машину, по его размеру в мегабайтах. Сначала мы воспользуемся заранее подготовленным набором данных (листинг 2.1), но, если хотите, можете создать аналогичный набор данных и смоделировать сетевую статистику собственной системы.

Листинг 2.1. Жестко «зашитые» обучающие данные и контрольные данные (из CodePen 2-a)

```
<script src='https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest'></script>
<script>
const trainData = {
```

[1] Цикл «чтение — оценка — вывод», известный также под названием интерактивного интерпретатора (командной оболочки). С помощью REPL мы можем активно взаимодействовать с кодом, запрашивать значения переменных и тестировать функции.

```
  sizeMB:  [0.080, 9.000, 0.001, 0.100, 8.000,
            5.000, 0.100, 6.000, 0.050, 0.500,
            0.002, 2.000, 0.005, 10.00, 0.010,
            7.000, 6.000, 5.000, 1.000, 1.000],
  timeSec: [0.135, 0.739, 0.067, 0.126, 0.646,
            0.435, 0.069, 0.497, 0.068, 0.116,
            0.070, 0.289, 0.076, 0.744, 0.083,
            0.560, 0.480, 0.399, 0.153, 0.149]
};
const testData = {
  sizeMB:  [5.000, 0.200, 0.001, 9.000, 0.002,
            0.020, 0.008, 4.000, 0.001, 1.000,
            0.005, 0.080, 0.800, 0.200, 0.050,
            7.000, 0.005, 0.002, 8.000, 0.008],
  timeSec: [0.425, 0.098, 0.052, 0.686, 0.066,
            0.078, 0.070, 0.375, 0.058, 0.136,
            0.052, 0.063, 0.183, 0.087, 0.066,
            0.558, 0.066, 0.068, 0.610, 0.057]
};
</script>
```

В предыдущем листинге HTML-кода мы явным образом указали теги `<script>`, чтобы продемонстрировать загрузку последней версии библиотеки TensorFlow.js с помощью суффикса `@latest` (на момент написания книги этот код работал с версией 0.13.5). Далее мы расскажем подробнее о различных способах импорта TensorFlow.js в приложение, но в дальнейшем теги `<script>` будут подразумеваться. Первый `<script>` загружает пакет TensorFlow и описывает символ `tf`, с помощью которого можно ссылаться на имена в этом пакете. Например, `tf.add()` ссылается на операцию TensorFlow для сложения двух тензоров. Забегая вперед: мы будем считать, что символ `tf` загружен и доступен в глобальном пространстве имен путем выполнения в текущей среде сценария TensorFlow.js, как показано выше.

В листинге 2.1 создаются две константы, `trainData` и `testData`, каждая из которых содержит по 20 примеров данных: длительности скачивания файла (`timeSec`) и размера этого файла (`sizeMB`). Элементы из `sizeMB` и `timeSec` взаимно однозначно соответствуют друг другу. Например, первый элемент `sizeMB` из `trainData` равен 0,08 Мбайт, и скачивание этого файла занимает 0,135 секунды — это первый элемент `timeSec` — и т. д. Наша цель в примере — оценить `timeSec` по заданному `sizeMB`. В первом примере мы формируем данные, просто «зашивая» их в наш код. Для такого простого примера этот подход целесообразен, но при росте размера набора данных код быстро станет очень громоздким. В дальнейших примерах мы покажем, как выполнять потоковую трансляцию данных из внешнего хранилища или по сети.

Вернемся к данным. Из графика на рис. 2.2 видно, что существует легко предсказуемая, пусть и не идеальная, зависимость между размером файла и временем скачивания. Данные на практике обычно зашумлены, но похоже, что можно произвести достаточно неплохую линейную оценку продолжительности скачивания по размеру файла. Если судить на глаз, продолжительность скачивания равна примерно 0,1 секунды при нулевом размере файла, а далее растет примерно на 0,07 секунды

на каждый дополнительный мегабайт. Как вы помните из главы 1, каждую пару вход/выход иногда называют *примером данных* (example). Выходной сигнал часто называют *целевой величиной/переменной/признаком* (target), а элементы входного сигнала обычно называют *входными признаками* (features). В нашем случае каждый из 40 примеров содержит ровно один признак, `sizeMB`, и числовую целевую величину, `timeSec`.

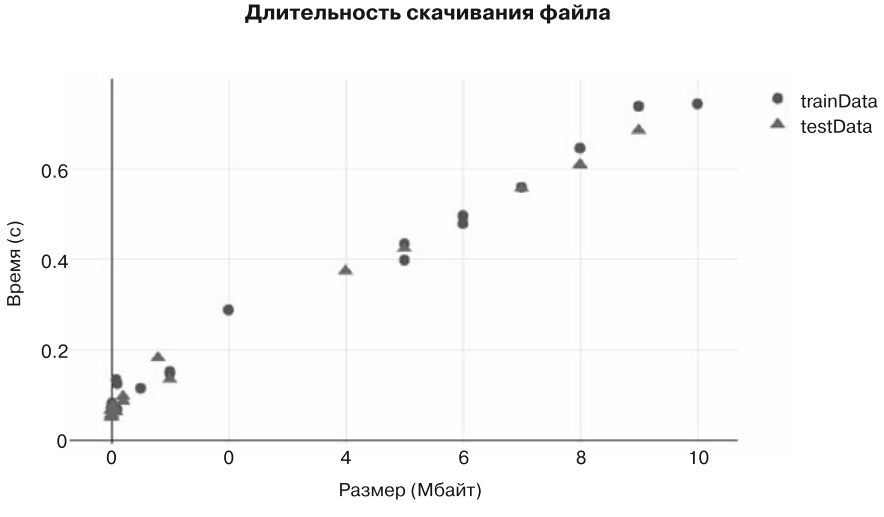


Рис. 2.2. Измеренная продолжительность скачивания относительно размера файла. Если вам интересно, как создавать подобные графики, то соответствующий код можно найти в codepen.io/tfjsbook/pen/dgQVze

Возможно, вы обратили внимание в листинге 2.1, что данные разбиты на два поднабора, а именно: `trainData` и `testData`. `trainData` — обучающий набор данных, содержит примеры данных, на которых будет обучаться модель. `testData` — контрольный набор данных. С его помощью мы будем определять, насколько хорошо обучена модель, по завершении обучения. Обучать и оценивать качество модели на одних и тех же данных — все равно что сдавать экзамен, заранее зная все правильные ответы. В предельном случае модель может теоретически запомнить значения `timeSec` для всех `sizeMB` в обучающих данных — не слишком удачный алгоритм обучения. В результате такая модель вряд ли смогла бы хорошо судить о будущих данных, ведь маловероятно, что значения будущих входных признаков окажутся точно такими же, как и те, на которых обучалась модель.

Следовательно, технологический процесс должен быть таким. Сначала мы обучаем нейронную сеть на обучающих данных производить точные предсказания `timeSec` по заданному `sizeMB`. Далее мы просим сеть сгенерировать предсказания для `sizeMB` на контрольных данных и измеряем, насколько близки эти предсказания к реальным значениям `timeSec`. Но сначала необходимо преобразовать эти данные в понятный TensorFlow.js формат, что и будет нашим первым примером использования тензоров. Код в листинге 2.2 демонстрирует первое в нашей книге использование функций из

пространства имен `tf.*`. В нем мы увидим методы преобразования типа неформатированных структур данных JavaScript в тензоры.

Листинг 2.2. Преобразование типа данных в тензоры (из CodePen 2-b)

```
const trainTensors = {
  sizeMB: tf.tensor2d(trainData.sizeMB, [20, 1]),
  timeSec: tf.tensor2d(trainData.timeSec, [20, 1])
};
const testTensors = {
  sizeMB: tf.tensor2d(testData.sizeMB, [20, 1]),
  timeSec: tf.tensor2d(testData.timeSec, [20, 1])
};
```

[20, 1] здесь — форма тензора. Подробнее мы расскажем позже, но эта форма означает, что мы хотим интерпретировать список чисел как 20 примеров данных, каждый из которых состоит из одного числа. Если форма очевидна из, скажем, структуры данных, то этот аргумент можно опустить

И хотя использовать эти API довольно просто, читателям, желающим лучше разобраться в них, стоит заглянуть в приложение Б. Оно охватывает не только функции создания тензоров наподобие `tf.tensor2d()`, но и функции для операций преобразования и объединения тензоров, а также паттерны удобной упаковки в тензоры часто встречающихся на практике типов данных, например изображений и видеофайлов. Мы не станем углубляться в низкоуровневые API в основном тексте, поскольку этот материал довольно скучен и не связан с конкретными примерами задач.

Вообще говоря, тензоры являются основными структурами данных всех современных систем машинного обучения. Они играют основополагающую роль в этой сфере — настолько основополагающую, что TensorFlow и TensorFlow.js были названы в их честь. Краткое напоминание из главы 1: по своей сути тензор представляет собой контейнер для данных — практически всегда числовых. Так что его можно считать контейнером для чисел. Вероятно, вы уже знакомы с векторами и матрицами — одно- и двумерными тензорами соответственно. Тензоры — это обобщение матриц на пространства произвольной размерности. Количество измерений и размер каждого из них называются *формой* (shape) тензора. Например, матрица 3×4 представляет собой тензор с формой `[3, 4]`. Вектор длиной 10 представляет собой одномерный тензор с формой `[10]`.

В контексте тензоров измерения часто называют *осями координат* (axis). В TensorFlow.js тензоры — часто используемое представление, благодаря которому компоненты могут обмениваться информацией и работать друг с другом, неважно, на CPU, GPU или другом аппаратном обеспечении. Когда понадобится, мы расскажем больше про тензоры и распространенные сценарии их использования, а пока продолжим наш проект с предсказаниями.

2.1.4. Описываем простую модель

В сфере глубокого обучения функция отображения входных признаков в целевые называется *моделью*. Функция модели на основе полученных признаков выполняет вычисления и выдает предсказания. Модель, которую мы создадим здесь, представ-

ляет собой функцию, принимающую на входе размер файла и выдающую значения продолжительности скачивания (см. рис. 2.2). В терминологии глубокого обучения *сеть* (network) иногда используется в качестве синонима слова «модель». Наша первая модель реализует *линейную регрессию* (linear regression).

Регрессия в контексте машинного обучения означает, что модель возвращает вещественные значения, пытаясь подобрать соответствие для целевых признаков. Этот метод отличен от классификации, при которой выходные сигналы выбираются из некоего набора вариантов. В задачах регрессии модель, выдающая более близкие к целевым значения, лучше, чем модель, выдающая более далекие. Лучше, если модель предсказывает, что скачивание файла размером 1 Мбайт занимает около 0,15 секунды, чем если бы она предсказывала, что оно займет 600 секунд.

Линейная регрессия — особый вид регрессии, при котором выходной сигнал как функция входного представляет собой прямую линию (или соответственно плоскость в многомерном пространстве, если входных признаков несколько) (листинг 2.3). Важную роль играет возможность *подстройки* моделей. Это означает возможность корректировки вычислений, переводящих входной сигнал в выходной. Мы воспользуемся этим свойством для подстройки модели так, чтобы она лучше «подходила» для наших данных. В линейном случае отношение входного сигнала к выходному сигналу модели всегда является прямой линией, но можно подбирать ее наклон и точку пересечения с осью координат *Y*.

Листинг 2.3. Формирование модели линейной регрессии (из CodePen 2-c)

```
const model = tf.sequential();
model.add(tf.layers.dense({inputShape: [1], units: 1}));
```

Основной кирпичик нейронных сетей — *слой* (layer), модуль обработки данных, который можно считать подстраиваемой функцией, переводящей тензоры в тензоры. В данном случае наша модель состоит из одного плотного слоя с наложенным на форму входного тензора ограничением, описанным с помощью параметра `inputShape: [1]`. Это значит, что слой ожидает входные данные в виде одномерного тензора, хранящего только одно значение. Выходной сигнал плотного слоя всегда представляет собой одномерный тензор для каждого примера данных, но размер этого измерения определяется конфигурационным параметром `units`. В данном случае нам нужно лишь одно выходное значение, поскольку мы хотим предсказать одно число, а именно `timeSec`.

По существу, плотный слой представляет собой настраиваемую функцию умножения-сложения, переводящую каждый входной сигнал в соответствующий выходной. А поскольку на входе и выходе только по одному значению, эта модель представляет собой простое линейное уравнение `y = m * x + b`, наверняка знакомое вам из школьного курса математики. В плотном слое `m` называется *ядром* (kernel), а `b` — *смещением* (bias), как показано на рис. 2.3. В данном случае мы сформировали линейную модель для отношения между входным (`sizeMB`) и выходным (`timeSec`) сигналами:

```
timeSec = ядро * sizeMB + смещение
```

В этом уравнении четыре члена. Два из них с точки зрения обучения модели фиксированы: значения `sizeMB` и `timeSec` определяются обучающими данными (см. листинг 2.1). Оставшиеся два члена, ядро и смещение, представляют собой параметры модели. Их значения выбираются случайным образом при создании модели. Полученное на основе этих случайных значений предсказание продолжительности скачивания, конечно, хорошим не будет. Чтобы получить адекватные предсказания, необходимо найти хорошие значения ядра и смещения путем обучения модели на данных. Этот поиск и представляет собой *процесс обучения*.

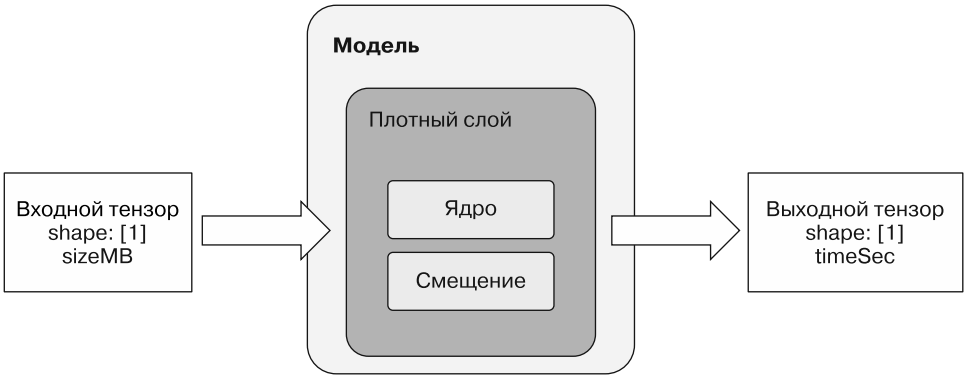


Рис. 2.3. Иллюстрация простой модели линейной регрессии. Модель включает только один слой. Подбираемые параметры модели (веса), ядро и смещение показаны внутри плотного слоя

Для поиска хороших значений ядра и смещения (которые совокупно называются *весами* или *весовыми коэффициентами*) необходимы две вещи:

- мера того, насколько хороши конкретные значения весов;
- способ обновления значений весов, чтобы на следующем шаге модель была лучше, чем на предыдущем, относительно упомянутой выше меры.

Это приводит нас к следующему шагу решения задачи линейной регрессии. Для подготовки сети к обучению необходимо выбрать меру и метод обновления, соответствующие двум нужным для модели элементам, перечисленным выше. Это делается на шаге, в терминологии TensorFlow.js, *компиляции модели* (model compilation), для которого нужны:

- *функция потерь* (loss function) — метрика погрешности. Именно на ее основе сеть измеряет качество работы на обучающих данных и определяет, в каком направлении двигаться дальше. При обучении требуется возможность построения графика функции потерь относительно времени. Эта функция должна стремиться к нулю. Если модель уже долго обучается, а потери не снижаются, вероятно, она не подгоняется к данным. В книге мы расскажем, как справиться с подобными проблемами;
- *оптимизатор* (optimizer) — алгоритм обновления сетью своих весов (в данном случае ядра и смещения) на основе данных и функции потерь.

В нескольких следующих главах мы основательно исследуем назначение функции потерь и оптимизатора и их выбор. А пока нам подойдут следующие (листинг 2.4).

Листинг 2.4. Настройка опций обучения: компиляция модели (из CodePen 2-c)

```
model.compile({optimizer: 'sgd', loss: 'meanAbsoluteError'});
```

Мы вызываем для модели метод `compile`, указывая в качестве оптимизатора `'sgd'`, а в качестве функции потерь — `'meanAbsoluteError'`. `'meanAbsoluteError'` означает, что наша функция потерь вычисляет абсолютное (положительное) значение удаленности предсказаний от целевых значений, после чего возвращает среднее значение полученного:

```
meanAbsoluteError = average( absolute(modelOutput - targets) )
```

Например, при:

```
modelOutput = [1.1, 2.2, 3.3, 3.6]
targets =     [1.0, 2.0, 3.0, 4.0]
```

получаем:

```
meanAbsoluteError = average([|1.1 - 1.0|, |2.2 - 2.0|,
                             |3.3 - 3.0|, |3.6 - 4.0|])

                  = average([0.1, 0.2, 0.3, 0.4])

                  = 0.25
```

При очень плохих предсказаниях (очень удаленных от целевых признаков) `meanAbsoluteError` будет очень большим. И напротив, в идеальном случае все предсказания будут совершенно точны, а разница между выходным сигналом нашей модели и целевыми значениями равна нулю, а значит, равна нулю и функция потерь (`meanAbsoluteError`).

`sgd` в листинге 2.4 означает *стохастический градиентный спуск* (stochastic gradient descent), который мы опишем чуть подробнее в разделе 2.2. Если вкратце, это значит, что для выбора корректировок весов, необходимых для снижения потерь, мы воспользуемся математическим анализом. После чего выполним эти корректировки и повторим процесс.

Наша модель готова, можно подгонять ее к обучающим данным.

2.1.5. Подгонка модели к обучающим данным

Обучение модели в TensorFlow.js запускается вызовом ее метода `fit()`. Происходит подгонка (fit) модели под обучающие данные. В данном случае мы передаем тензор `sizeMB` в качестве входного сигнала и тензор `timeSec` в качестве желаемого выходного. Мы также передаем объект с настройками конфигурации, содержащий поле

epochs, означающее, что мы хотели бы пройти по обучающим данным ровно десять раз. В глубоком обучении отдельный проход по полному обучающему набору данных называется *эпохой* (epoch).

Листинг 2.5. Подгонка модели линейной регрессии (из Codepen 2-c)

```
(async function() {
  await model.fit(trainTensors.sizeMB,
                  trainTensors.timeSec,
                  {epochs: 10});
})();
```

Метод `fit()` может выполняться довольно долго, секунды или даже минуты. Поэтому мы воспользовались возможностью *async/await* стандарта ES2017/ES8, чтобы эта функция не блокировала основной поток выполнения UI при работе в браузере. Это аналогично другим «долгоиграющим» функциям JavaScript, например `async fetch`. Здесь мы ожидаем завершения вызова `fit()`, прежде чем продолжать выполнение, с помощью паттерна *«Немедленно выполняемое асинхронное функциональное выражение»* (Immediately Invoked Async Function Expression)[1]. Но в будущих примерах обучение будет происходить в фоновом режиме, в то время как остальная работа будет выполняться в приоритетном потоке.

По завершении подгонки модели хочется проверить, как она работает. Главное, проверять модель необходимо на данных, которые не использовались во время обучения. Вопрос отделения контрольных данных от обучающих (которое позволяет избежать обучения на контрольных данных) будет проходить красной нитью по всей книге. Желательно как следует усвоить эту важнейшую составляющую технологического процесса машинного обучения.

Метод `evaluate()` модели вычисляет функцию потерь для переданных ему признаков и целевых значений примера данных. Он напоминает метод `fit()` тем, что вычисляет ту же функцию потерь, но отличается тем, что не обновляет веса модели. Мы используем `evaluate()` для оценки работы модели на контрольных данных, то есть чтобы понять, насколько хорошо она будет работать в дальнейшем:

```
> model.evaluate(testTensors.sizeMB, testTensors.timeSec).print();
Tensor
    0.31778740882873535
```

Здесь видно, что потери, усредненные по контрольным данным, составляют около 0,318. Конечно, поскольку модели обучаются со случайного начального состояния, вы получите другое значение. Еще один способ выразить ту же мысль: средняя абсолютная погрешность (mean absolute error, MAE) этой модели составляет более 0,3 секунды. Хорошо ли это? Лучше ли, чем константная оценка? Одна неплохая возможная константа для этого — средняя задержка. Подсчитаем, какую погрешность это нам даст, воспользовавшись поддержкой математических операций над тензорами библиотеки TensorFlow.js. Во-первых, мы вычислим среднее время скачивания для нашего обучающего набора данных:

---

[1] Больше информации о паттерне вы найдете по адресу http://mng.bz/RPOZ.

```
> const avgDelaySec = tf.mean(trainData.timeSec);
> avgDelaySec.print();
Tensor
    0.2950500249862671
```

Далее вычислим `meanAbsoluteError` вручную. MAE представляет собой просто среднюю удаленность нашего предсказания от реального значения. Мы воспользуемся `tf.sub()` для вычисления разницы между целевыми значениями и нашим (константным) предсказанием, а также `tf.abs()` для вычисления абсолютного значения (поскольку иногда оно оказывается меньше истинного, а иногда — больше), после чего вычислим среднее значение с помощью `tf.mean()`:

```
> tf.mean(tf.abs(tf.sub(testData.timeSec, 0.295))).print();
Tensor
    0.22020000219345093
```

О том, как вычислить то же самое с помощью лаконичного цепочечного API, читайте в инфобоксе 2.1.

ИНФОБОКС 2.1. Цепочечный API для тензоров

Помимо стандартного API, в котором функции работы с тензорами доступны в пространстве имен `tf`, к большинству функций работы с тензорами можно обратиться и из самих тензорных объектов, что позволяет при желании соединять подобные операции цепочкой. Следующий код функционально идентичен вычислению `meanAbsoluteError`, приведенному выше в основном тексте.

```
// Паттерн цепочечного API
> testData.timeSec.sub(0.295).abs().mean().print();
Tensor
    0.22020000219345093
```

Похоже, что средняя задержка составляет примерно 0,295 секунды и, если всегда предсказывать среднее значение, оценка получается лучше, чем у нашей сети. Это значит, что качество работы нашей модели хуже, чем у тривиального подхода, основанного просто на здравом смысле! Можно ли улучшить ее? Возможно, мы просто обучались в течение недостаточного количества эпох. Напоминаем, что при обучении значения ядра и смещения обновляются пошагово. В данном случае каждая эпоха соответствует одному шагу. Если модель обучалась лишь на протяжении небольшого количества эпох, значения параметров могли просто не успеть стать оптимальными. Давайте обучим нашу модель в течение еще нескольких эпох и снова оценим результат:

```
>  model.fit(trainTensors.sizeMB,
             trainTensors.timeSec,
             {epochs: 200});
```

Необходимо дождаться разрешения промиса, возвращаемого model.fit, прежде чем выполнять model.evaluate

```
>  model.evaluate(testTensors.sizeMB, testTensors.timeSec).print();
Tensor
    0.04879039153456688
```

Намного лучше! Похоже, что ранее мы *недообучили* (underfit) модель, то есть она недостаточно приспособилась к обучающим данным. Теперь наши оценки в среднем в пределах 0,05 секунды. Точность — в четыре раза выше, чем у «наивного» подхода со средним значением. В книге мы расскажем, как избежать недообучения, а равно и более хитрой проблемы *переобучения* (overfitting), при которой модель *слишком хорошо* подгоняется к обучающим данным и плохо обобщается на новые, не виденные ею данные.

2.1.6. Используем обученную модель для предсказаний

Замечательно! У нас есть модель, способная достаточно точно предсказывать время скачивания по размеру файла, но как ее использовать? Ответ: воспользоваться ее методом `predict()`:

```
> const smallFileMB = 1;
> const bigFileMB = 100;
> const hugeFileMB = 10000;
> model.predict(tf.tensor2d([[smallFileMB], [bigFileMB],
    [hugeFileMB]])).print();
Tensor
    [[0.1373825  ],
     [7.2438402  ],
     [717.8896484]]
```

Как видим, модель предсказывает, что скачивание файла размером 10 000 Мбайт займет около 718 секунд. Обратите внимание, что в наших обучающих данных не было примеров с подобным размером файла. В целом экстраполяция на значения, сильно выходящие за пределы обучающих данных, — рискованная затея, но в случае настолько простой задачи предсказание может оказаться точным... если не возникнут новые проблемы с буферами памяти, связью ввода-вывода и т. д. Лучше бы собрать побольше обучающих данных в этом диапазоне.

Необходимо также обернуть входные переменные в тензор соответствующей формы. В листинге 2.3 мы описали `inputShape` с формой `[1]`, так что модель ожидает примеры данных именно такой формы. Оба метода, `fit()` и `predict()`, работают сразу с несколькими примерами данных за раз. Для передачи `n` примеров данных мы упаковываем их в один входной тензор, форма которого, таким образом, должна быть `[n, 1]`. Если же мы перепутали и передали в модель вместо этого тензор с неправильной формой, то будет возвращена ошибка неправильной формы, как в следующем коде:

```
> model.predict(tf.tensor1d([smallFileMB, bigFileMB, hugeFileMB])).print();
Uncaught Error: Error when checking : expected dense_Dense1_input to have 2
dimension(s), but got array with shape [3]
```

Остерегайтесь подобных нестыковок формы, это очень распространенный тип ошибок!

2.1.7. Резюме нашего первого примера

Для такого маленького примера можно наглядно показать результаты модели. На рис. 2.4 построен график выходного сигнала модели (`timeSec`) как функции входного сигнала (`sizeMB`) для моделей в четырех различных точках процесса, начиная с недообученной модели после десяти эпох и заканчивая полностью сошедшейся моделью. Как видим, сошедшаяся модель хорошо подогнана к данным. Если вы хотели бы научиться строить графики, подобные приведенному на рис. 2.4, загляните в CodePen (codepen.io/tfjs-book/pen/VEVMMd).

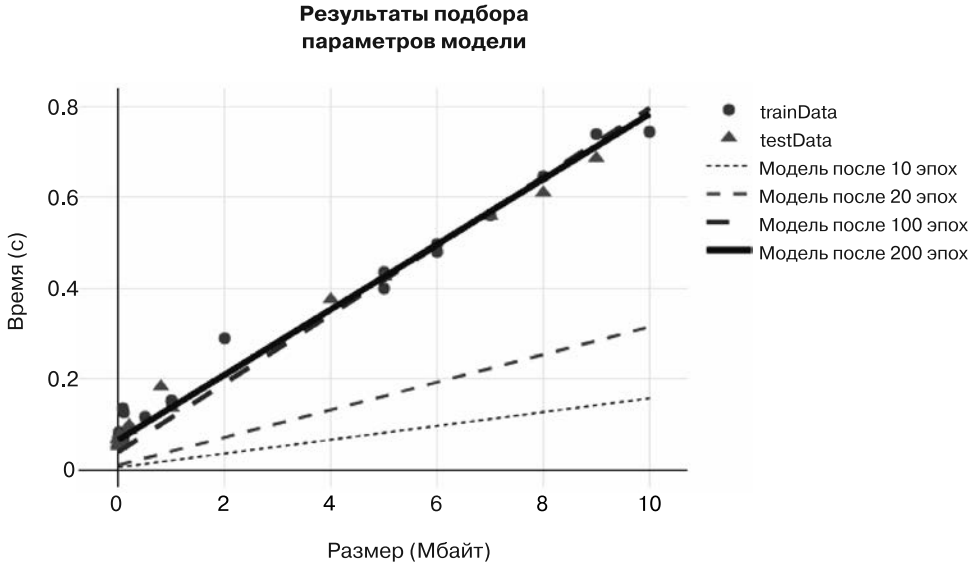


Рис. 2.4. Подобранная линейная модель после обучения в течение 10, 20, 100 и 200 эпох

На этом наш первый пример завершается. Вы только что видели, как создать, обучить и оценить эффективность модели TensorFlow.js с помощью всего нескольких строк кода на JavaScript (листинг 2.6). В следующем разделе мы немного подробнее поговорим о происходящем внутри метода `model.fit`.

Листинг 2.6. Модель: описание, обучение, оценка и предсказание

```
const model = tf.sequential([tf.layers.dense({inputShape: [1], units: 1})]);
model.compile({optimizer: 'sgd', loss: 'meanAbsoluteError'});
(async () => await model.fit(trainTensors.sizeMB,
                             trainTensors.timeSec,
                             {epochs: 10}))();
model.evaluate(testTensors.sizeMB, testTensors.timeSec);
model.predict(tf.tensor2d([[7.8]])).print();
```

2.2. Внутреннее устройство Model.fit(): анализируем градиентный спуск из примера 1

В предыдущем разделе мы создали простую модель и подогнали ее к обучающим данным, показав, что по размеру файла можно выполнять достаточно точные предсказания продолжительности скачивания. Это не самая впечатляющая нейронная сеть, но она функционирует совершенно аналогично бóльшим, значительно более сложным системам, которые мы будем создавать в дальнейшем. Мы видели, что подгонка ее в течение десяти эпох не дала хороших результатов, а вот в результате подгонки в течение 200 эпох получилась качественная модель[1]. Немного углубимся в подробности и разберемся, что именно происходит «под капотом» во время обучения модели.

2.2.1. Основные идеи оптимизации на основе градиентного спуска

Напомним, что наша простая, однослойная модель подгоняет линейную функцию `f(входной_сигнал)` вида:

```
выходной_сигнал = ядро * входной_сигнал + смещение
```

где ядро и смещение — подбираемые параметры (весá) плотного слоя. Эти веса содержат информацию, усвоенную сетью из обучающих данных.

Изначально эти веса содержат маленькие случайные значения (шаг под названием *«задание случайных начальных значений»*). Конечно, бессмысленно ожидать чего-то полезного от величины `ядро * входной_сигнал + смещение`, если `ядро` и `смещение` случайные. Представьте себе, как значение MAE меняется при выборе различных значений этих параметров. Можно ожидать, что значение функции потерь будет невелико, если они близки к угловому коэффициенту и точке пересечения с осью координат прямой на рис. 2.4, и станет ухудшаться, когда эти параметры будут описывать совсем другие прямые. Эта идея — потери как функция всех подбираемых параметров — известна под названием *«поверхность ошибок»* (loss surface).

Поскольку это лишь крошечный пример всего с двумя подбираемыми параметрами и одной целевой переменной, мы можем изобразить поверхность ошибок в виде двумерного контурного графика, как демонстрирует рис. 2.5. Эта поверхность ошибок имеет аккуратную чашеобразную форму с глобальным минимумом внизу чаши, соответствующим оптимальным значениям параметров. Обычно, впрочем, поверхность ошибок моделей глубокого обучения намного сложнее. Она может насчитывать гораздо больше двух измерений и множество локальных минимумов, то есть точек, которые ниже окружающей их части поверхности, но не всей поверхности в целом.

---

[1] Обратите внимание, что для таких простых линейных моделей существуют простые, эффективные аналитические решения. Впрочем, приведенный метод оптимизации работает даже для более сложных моделей, которые мы рассмотрим позднее.

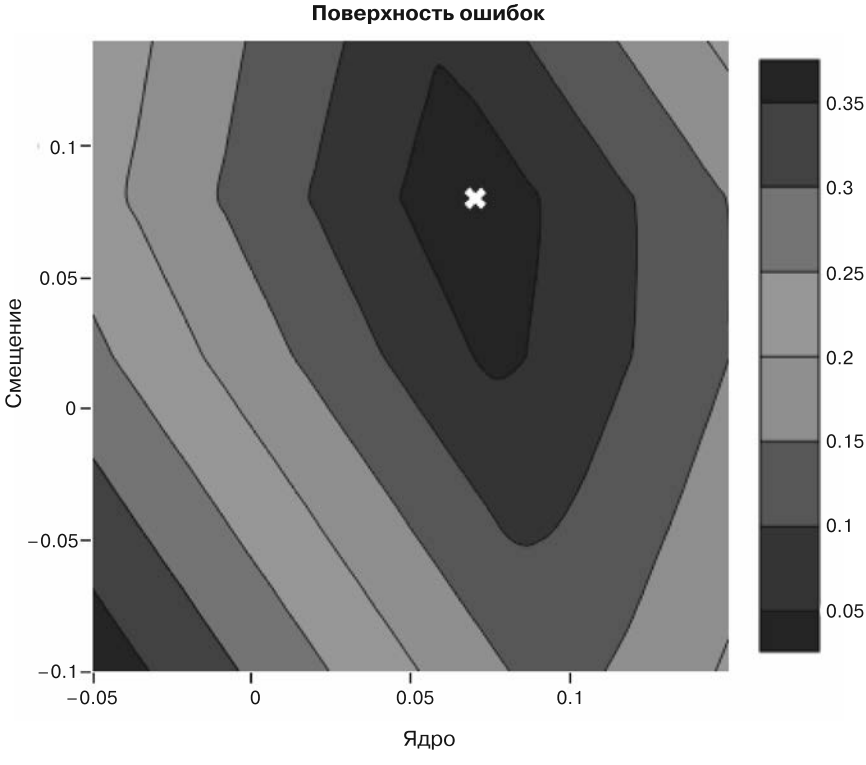


Рис. 2.5. Поверхность ошибок иллюстрирует зависимость функции потерь от настраиваемых параметров модели в виде контурного графика. С высоты птичьего полета представляется, что достаточно хорошим вариантом низкого значения потерь будет точка {смещение: 0.08, ядро: 0.07}, отмеченная белым крестиком. Очень редко удается проверить все значения параметров и построить подобный график. В подобном случае оптимизация сильно упрощается: достаточно выбрать параметры, соответствующие минимальному значению функции потерь!

Мы видим, что эта поверхность ошибок имеет форму чаши, а наилучшее (минимальное) значение находится в районе точки `{смещение: 0.08, ядро: 0.07}`. Это соответствует геометрии прямой, вытекающей из наших данных, с временем скачивания около 0,1 секунды, даже если размер файла близок к нулю. Инициализация модели случайными значениями запускает процесс при случайных параметрах, подобно случайной точке на карте, для которой вычисляются начальные потери. Далее мы постепенно подстраиваем параметры на основе сигнала обратной связи. Эта постепенная подстройка и есть *обучение* в смысле машинного обучения. Оно происходит внутри *цикла обучения* (training loop), приведенного на рис. 2.6.

Рисунок 2.6 демонстрирует итерации цикла обучения по этим шагам (выполняемые столько раз, сколько нужно).

1. Выбираем *батч* обучающих примеров данных `x` и соответствующие целевые значения `y_true`. Батч — это просто набор входных примеров данных в виде тензора. Количество примеров данных в батче называется его *размером*. На практике при глубоком обучении в качестве размера часто выбирается степень двойки,

например 128 или 256. Примеры данных собираются в батчи, чтобы полнее использовать возможности параллельных вычислений GPU и обеспечить устойчивость вычисляемых значений градиентов.

2. Получаем предсказания `y_pred`, применяя сеть к батчу `x` (этот шаг носит название *прямого прохода* (forward pass)).
3. Вычисляем функцию потерь сети на данном батче — меру расхождения между `y_pred` и `y_true`.
4. Обновляем все веса (параметры) сети таким образом, чтобы слегка уменьшить потери на этом батче. Конкретные изменения отдельных весов производит оптимизатор — еще одна опция вызова `model.compile()`.

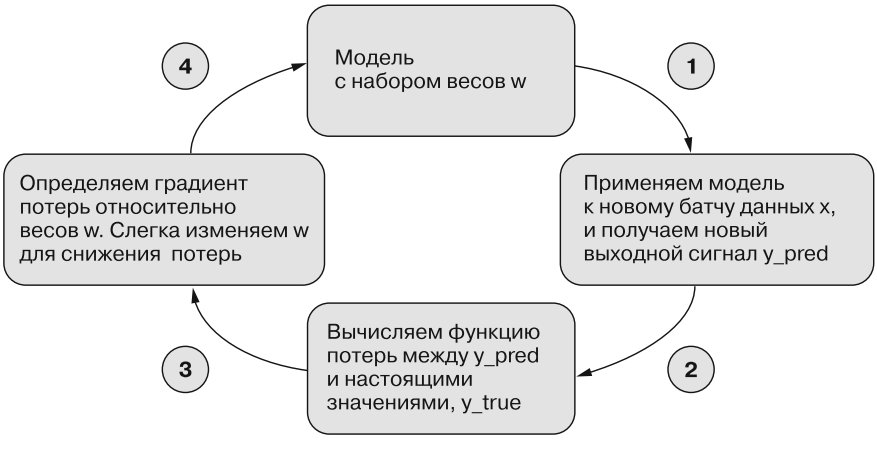


Рис. 2.6. Блок-схема, иллюстрирующая цикл обучения с обновлением модели путем градиентного спуска

Если уменьшать потери на каждом шаге, в конце концов мы получим сеть с низким значением функции потерь на обучающих данных. Сеть научилась отображать входные сигналы в правильные целевые значения. Со стороны это может показаться каким-то волшебством, но все становится очень простым, если разбить на элементарные шаги.

Единственная хитрость заключается в шаге 4: как определить, какие веса увеличить, какие уменьшить и насколько? Можно просто высказывать догадки и проверять их, принимая только те обновления, которые действительно уменьшают потери. Такой алгоритм может подойти для простой задачи, как вышеупомянутая, но будет работать очень медленно. Для более серьезных задач, при оптимизации миллионов весов, вероятность случайного выбора хорошего направления ничтожно мала. Лучше будет воспользоваться фактом *дифференцируемости* всех используемых в сети операций и вычислить *градиент* функции потерь относительно параметров сети.

Что такое градиент? Не будем давать формальное определение (с использованием теории математического анализа), а опишем его на интуитивном уровне: *«Направление, при крошечном изменении всех весов в котором функция потерь растет быстрее всего среди всех направлений»*.

И хотя это определение не слишком формально, здесь есть над чем подумать, так что рассмотрим его по частям.

- Прежде всего, градиент — вектор, количество элементов в котором совпадает с количеством весов. Он отражает одно из направлений в пространстве всех возможных выборов значений весов. Если веса модели состоят из двух чисел, как в случае нашей простой сети линейной регрессии, то градиент — двумерный вектор. Модели глубокого обучения часто насчитывают тысячи или миллионы измерений, так что градиенты этих моделей представляют собой векторы (направления) из тысяч или миллионов элементов.
- Во-вторых, градиент зависит от текущих значений весов. Другими словами, при различных значениях весов получаются различные градиенты. Это ясно из рис. 2.5, в котором направление наиболее быстрого спуска зависит от текущего положения на поверхности ошибок. На левом краю необходимо идти направо. Около дна следует двигаться вверх и т. д.
- Наконец, математическое определение градиента говорит о направлении, вдоль которого функция потерь *растет*. Разумеется, при обучении нейронных сетей нам нужно, чтобы функция потерь *падала*. Именно поэтому нам нужно менять веса в направлении, *противоположном* градиенту.

В качестве аналогии рассмотрим прогулку по горам. Допустим, мы хотели бы идти по местности с наименьшей высотой над уровнем моря. В этой аналогии мы можем менять высоту над уровнем моря, перемещаясь в любом направлении, задаваемом осями координат запад — восток и север — юг. Первый пункт вышеприведенного списка можно интерпретировать как то, что градиент нашей высоты над уровнем моря представляет собой наиболее крутое направление относительно текущего наклона почвы под нашими ногами. Второй пункт очевиден: он гласит, что наиболее крутое направление зависит от нашего текущего местоположения. Наконец, чтобы попасть на расположенную ниже всего над уровнем моря местность, необходимо двигаться в направлении, *противоположном* градиенту.

Данный процесс обучения вполне логично называют *градиентным спуском* (gradient descent). Помните, в листинге 2.4 мы задавали оптимизатор модели с помощью опции `optimizer: 'sgd'`? Теперь вам должна быть понятна относящаяся к градиентному спуску часть названия «стохастический градиентный спуск». А «стохастический» означает просто выбор на каждом шаге градиентного спуска случайных примеров данных из обучающего набора вместо использования всех обучающих примеров на каждом шаге. Стохастический градиентный спуск — всего лишь модификация градиентного спуска, используемая для большей эффективности вычислений.

Теперь у нас есть все необходимое, чтобы объяснить, как работает механизм оптимизации и почему 200 эпох лучше, чем десять, для нашей модели времени скачивания. На рис. 2.7 показано, как алгоритм градиентного спуска следует вниз по поверхности ошибок в поисках весовых коэффициентов, лучше всего подходящих для наших обучающих данных. Контурный график из блока A на рис. 2.7 демонстрирует ту же поверхность ошибок, что и раньше, в несколько меньшем масштабе, с наложенным на нее путем, по которому движется алгоритм градиентного спуска. Этот путь начинается со *случайной инициализации* — случайного места на рисунке. Нам нужно выбрать какое-то случайное место, с которого начинать, ведь заранее мы не знаем

оптимального места. По дороге мы обнаруживаем еще несколько интересных мест, соответствующих недообученным и хорошо обученным моделям. В блоке Б рис. 2.7 показан график потерь модели как функции номера шага и отмечены аналогичные интересные места. Блок В иллюстрирует модели с помощью весов в качестве срезов состояния на показанных в блоке Б шагах.

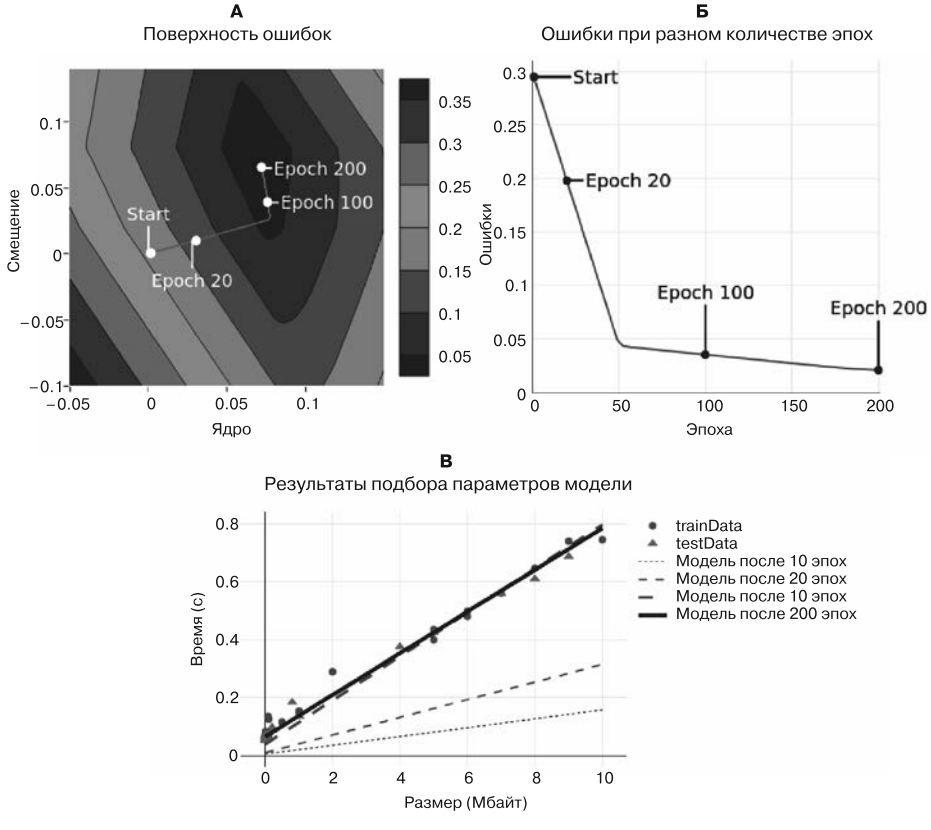


Рис. 2.7. Блок А: 200 не очень больших шагов на основе градиентного спуска приводят значения параметров в глобальный экстремум. На рисунке отмечены начальные значения весов и их значения после 20, 100 и 200 эпох. Блок Б: график потерь как функции от номера эпохи с отмеченными значениями потерь в тех же точках. Блок В: функция зависимости timeSec от sizeMB, воплощаемая обученной моделью после 10, 20, 100 и 200 эпох, повторенных здесь для удобства сравнения местоположения на поверхности ошибок и результата работы модели. Поэкспериментировать с этим кодом можно на codepen.io/tfjs-book/pen/JmerMM

Наша простая модель линейной регрессии — единственная в этой книге, где мы можем позволить себе роскошь столь наглядной визуализации процесса градиентного спуска. Но, когда позже мы начнем рассматривать более сложные модели, учтите, что сущность градиентного спуска остается той же самой: просто итеративный спуск по склону сложной, многомерной поверхности в надежде, что рано или поздно мы попадем в точку с очень низким значением функции потерь.

Изначально мы использовали размер шага по умолчанию (определяемый *скоростью обучения по умолчанию* (default learning rate)), но всего десяти шагов прохода по нашему ограниченному набору данных оказалось недостаточно для достижения оптимального решения; 200 шагов же вполне достаточно.

Вообще говоря, как узнать, какую скорость обучения выбрать и когда его завершать? Существует несколько удобных эмпирических правил, которые мы будем обсуждать по ходу книги, но универсального правила на все случаи жизни нет. Если скорость обучения слишком мала, а значит, используется *слишком маленький* шаг, достичь оптимальных параметров за приемлемое время не получится. И напротив, при слишком большой скорости обучения, а значит, и слишком *большом* размере шага мы можем «проскочить» минимум и получить решение даже с бóльшим значением функции потерь, чем то, с которого начали. В результате параметры нашей модели будут сильно колебаться около оптимальных, вместо того чтобы быстро и непринужденно приближаться к ним. На рис. 2.8 показано, что происходит, когда шаг градиента слишком велик. В еще более экстремальных случаях, при большой скорости обучения, значения параметров расходятся и стремятся к бесконечности, приводя к возникновению значений `NaN` (not-a-number — «не число») весов, и полностью портят модель.

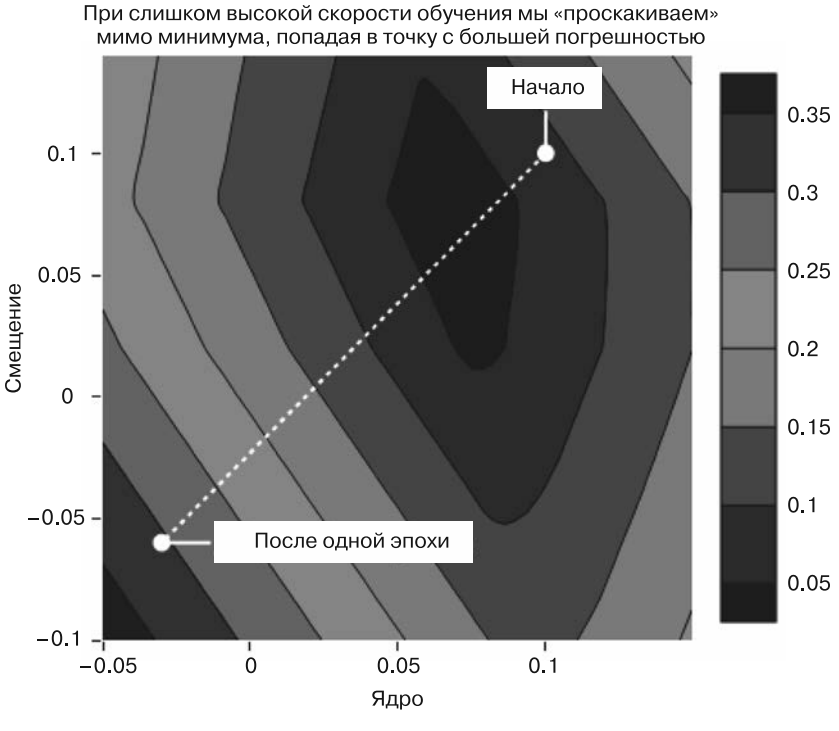


Рис. 2.8. При очень большой скорости обучения шаг градиента слишком велик и новые значения параметров могут оказаться хуже старых. Результатом могут стать колебания значений или какой-либо другой вид неустойчивости, что приведет к бесконечным значениям параметров (NaN). Можете попробовать увеличить скорость обучения в коде из CodePen до 0,5, чтобы посмотреть на такое поведение

2.2.2. Обратное распространение ошибки: внутри градиентного спуска

В предыдущем разделе мы объяснили, как размер шага обновлений весов влияет на процесс градиентного спуска. Однако мы не обсуждали, как вычисляются *направления* обновлений. Они жизненно важны для процесса обучения нейронной сети и определяются градиентами относительно весов, а для вычисления градиентов используется метод *обратного распространения ошибки* (backpropagation). Изобретенный в 1960-х годах метод обратного распространения ошибки сегодня один из фундаментов нейронных сетей и глубокого обучения. В этом подразделе на простом примере мы разберем, как работает метод обратного распространения ошибки. Обратите внимание, что этот подраздел предназначен только для тех читателей, которые хотели бы лучше разобраться в обратном распространении ошибки. Читать его не обязательно, если вы хотите просто применить алгоритм с помощью TensorFlow.js, ведь все эти механизмы аккуратно спрятаны «под капотом» API `tf.Model.fit()`. Вы можете пропустить подраздел и продолжить чтение с раздела 2.3.

Рассмотрим простую линейную модель:

```
y' = v * x,
```

где `x` — входной признак, `y'` — предсказываемый выходной сигнал, а `v` — единственный весовой параметр этой модели, обновляемый в ходе обратного распространения ошибки. Пусть роль функции потерь у нас играет квадратичная погрешность; тогда `loss` (функция потерь), `v`, `x` и `y` (фактическое целевое значение) связаны таким соотношением:

```
ф_потерь = (y' - y)² = (v * x - y)²
```

Возьмем следующие конкретные значения: два входных `x = 2` и `y = 5`, а весовой коэффициент `v = 0`. Значение `loss` тогда равно `25`. Это все показано пошагово на рис. 2.9. Каждый серый прямоугольник в блоке А соответствует входному сигналу (то есть `x` и `y`). А каждый белый прямоугольник обозначает какую-либо операцию. Всего операций три. Соединяющие операции ребра (а также ребро, соединяющее подбираемый весовой коэффициент `v` с первой операцией) помечены `e1`, `e2` и `e3`.

Один из важных этапов обратного распространения ошибки состоит в определении следующей величины: *«Если все остальное (в данном случае величины x и y) не меняется, насколько изменится значение функции потерь, если поменять v на одну единицу измерения?»*

Эта величина называется *градиентом функции потерь по (относительно)* `v`. Зачем нам нужен этот градиент? Поскольку далее можно изменить `v` в *противоположном* ему направлении, в результате чего значение функции потерь уменьшится. Обратите внимание, что градиенты потерь по `x` или `y` нам не нужны, так как `x` и `y` обновлять не требуется: это фиксированные входные данные.

Этот градиент вычисляется пошагово, начиная со значения функции потерь и обратно к переменной `v`, как показано в блоке Б на рис. 2.9. Именно направление, в котором на этом рисунке выполняются вычисления, и служит причиной того, что алгоритм называется «обратное распространение ошибки». Рассмотрим его пошагово.

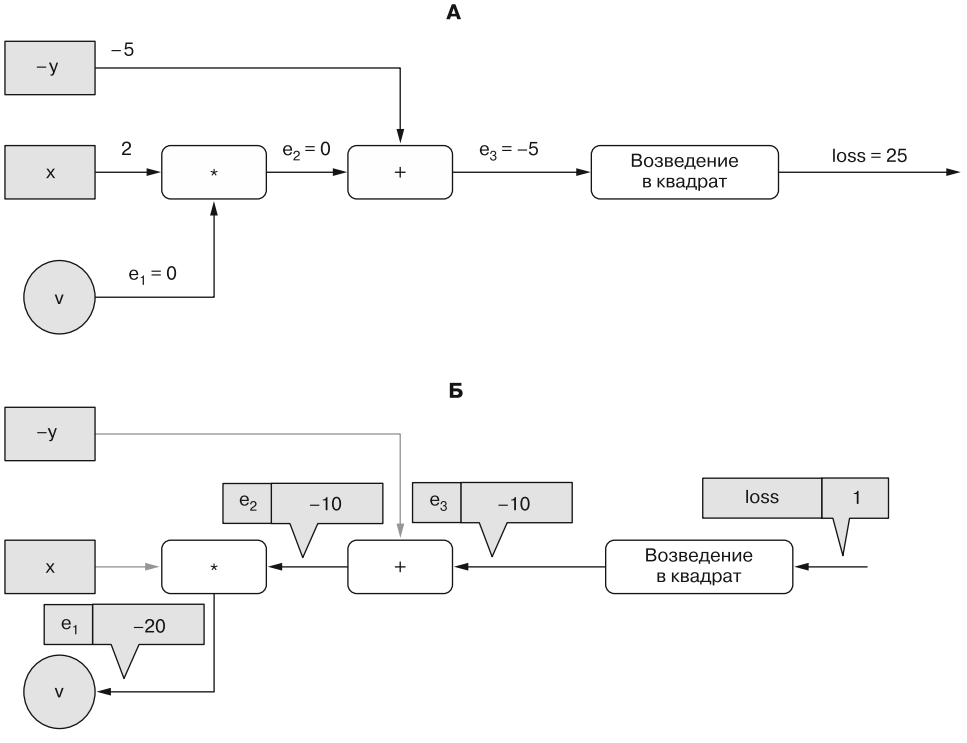


Рис. 2.9. Иллюстрация алгоритма обратного распространения ошибки на простой линейной модели со всего одним подбираемым весовым коэффициентом (v). Блок А: прямой проход модели — значение функции потерь вычисляется на основе веса (v) и входных сигналов (x и y). Блок Б: обратный проход — пошагово вычисляется градиент функции потерь по v, от потерь к v

- На ребре, обозначенном как `loss`, мы начинаем с градиента, равного 1. Это соответствует тривиальному утверждению: «единичный прирост `ф_потерь` соответствует единичному приросту самих потерь»[1].
- На помеченном e_3 ребре вычисляется градиент функции потерь относительно единичного изменения текущего значения e_3. А поскольку промежуточная операция представляет собой возведение в квадрат и поскольку из основ математического анализа известно, что производная (градиент в случае функции одной переменной) $(e_3)^2$ по e_3 равна `2 * e3`, то значение градиента равно `2 * -5 = -10`. Значение `-10` умножается на предыдущий градиент (то есть 1) и получается градиент на ребре e_3: `-10`. Именно таков прирост потерь при увеличении e_3 на 1. Как вы могли заметить, для получения из градиента потерь по одному ребру градиента потерь по следующему ребру необходимо умножить предыдущий градиент на

[1] На каждом ребре вычисляется градиент (частная производная) величины с предыдущего ребра относительно величины на этом ребре. Так как предыдущего ребра нет, вычисляется градиент потерь относительно потерь, который равен единице. В результате и получается тривиальное утверждение, что отмечает сам автор. — *Примеч. науч. ред.*

градиент, вычисленный локально в текущем узле. Это правило иногда называют *цепным правилом* (chain rule)[1].

- На ребре e_2 мы вычисляем градиент e_3 по e_2. А поскольку это простая операция сложения, то градиент равен просто 1, вне зависимости от второго входного значения (`-y`). Умножая эту 1 на градиент на ребре e_3, мы получаем градиент на ребре e_2, то есть –10.
- На ребре `e1` мы вычисляем градиент `e2` по `e1`. Выполняемая здесь операция — `x * v`. Значит, градиент e_2 по e_1 (то есть по `v`) равен `x`, то есть 2. Умножаем значение 2 на градиент по ребру e_2 и получаем итоговый градиент: `2 * -10 = -20`.

Пока мы получили градиент функции потерь по `v`, равный –20. Для градиентного спуска нам нужно умножить отрицательное значение этого градиента на скорость обучения. Пусть скорость обучения равна 0,01. Тогда получаем изменение градиента на:

```
-(-20) * 0.01 = 0.2
```

Именно на такую величину мы изменим `v` на этом шаге обучения:

```
v = 0 + 0.2 = 0.2
```

Поскольку `x = 2` и `y = 5`, а функция, параметры которой необходимо подобрать, — `y' = v * x`, то оптимальное значение `v` равно `5/2 = 2.5`. После первого шага обучения значение `v` меняется с 0 до 0,2. Другими словами, весовой коэффициент `v` чуть-чуть приближается к желаемому значению. На последующих шагах обучения он будет постепенно приближаться еще больше (игнорируя шум в обучающих данных) на основе описанного выше алгоритма обратного распространения ошибки.

Мы специально взяли такой простой пример, чтобы вам проще было следить за его ходом. И хотя в этом примере можно уловить основную суть метода обратного распространения ошибки, происходящее в настоящих нейронных сетях обратное распространение ошибки несколько отличается следующими нюансами.

- Вместо одного простого обучающего примера данных (в нашем случае `x = 2` и `y = 5`) обычно передается сразу батч из множества входных примеров данных. А используемое для вычисления градиента значение функции потерь представляет собой арифметическое среднее значений функции потерь для всех отдельных примеров.
- Обновляемые переменные обычно состоят из намного большего количества элементов. Так, вместо простой производной по одной переменной часто применяется матричное дифференциальное исчисление.
- Вместо вычисления градиента только по одной переменной обычно проводятся вычисления для нескольких переменных. На рис. 2.10 приведен пример, представляющий собой несколько более сложную линейную модель с двумя оптимизируемыми переменными. Помимо `k`, эта модель содержит еще и член для смещения: `y' = k * x + b`. Здесь приходится вычислять два градиента: один для `k`, а второй для `b`. Оба пути обратного распространения ошибки начинаются с `loss`, содержат часть общих ребер и формируют древовидную структуру.

[1] Или просто правилом дифференцирования сложной функции. — *Примеч. пер.*

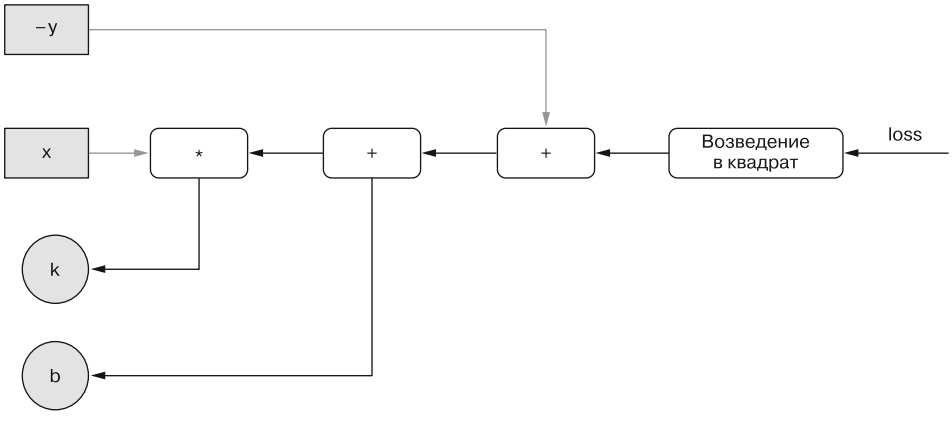


Рис. 2.10. Схематическое изображение обратного распространения ошибки от loss до двух обновляемых весовых коэффициентов (k и b)

Мы изложили здесь метод обратного распространения ошибки в общих чертах и довольно неформально. Чтобы глубже разобраться в математике и алгоритмах обратного распространения ошибки, загляните в приведенные в инфобоксе 2.2 библиографические ссылки.

Вы уже должны неплохо понимать, что происходит при подгонке простой модели к обучающим данным, так что отложим нашу крошечную задачу с предсказанием времени скачивания и воспользуемся TensorFlow.js для чего-нибудь более серьезного.

В следующем разделе мы создадим модель для точного предсказания цен недвижимости на основе сразу нескольких входных признаков.

ИНФОБОКС 2.2. Рекомендуемая литература по методу градиентного спуска и обратному распространению ошибки

Используемые для оптимизации нейронных сетей методы дифференциального исчисления очень интересны и позволяют лучше понять, как эти алгоритмы себя ведут. Но, за исключением самых основ, их вовсе *не обязательно* знать для практического применения машинного обучения, точно так же, как знания нюансов протокола TCP/IP полезны, но при создании современного веб-приложения без них можно обойтись. Мы приглашаем любознательных читателей заглянуть в перечисленные ниже замечательные источники информации, чтобы лучше разобраться в математических основах оптимизации сетей на основе градиентного спуска.

- Заметки о методе обратного распространения ошибки из лекции 4 курса CS231 Стэнфордского университета: http://cs231n.github.io/optimization-2/.
- *Hacker's Guide to Neural Nets* Андрея Карпаты (Andrej Karpathy): http://karpathy.github.io/neuralnets/.

2.3. Множественная линейная регрессия

В первом примере мы предсказывали наш целевой признак `timeSec` на основе лишь одного входного признака, `sizeMB`. Намного чаще встречается сценарий с несколькими входными признаками, когда неизвестно, какие из них наиболее полезны для предсказаний, а какие очень слабо связаны с целевыми. В таком случае приходится использовать их все одновременно, позволяя самому алгоритму обучения их рассортировать. В этом разделе мы и займемся подобной более сложной работой.

К концу этого раздела вы научитесь:

- создавать модели, получающие и усваивающие несколько входных признаков;
- создавать и выполнять веб-приложения с машинным обучением на основе Yarn, Git и стандартной структуры компоновки проектов JavaScript;
- обеспечивать большую устойчивость обучающих данных с помощью нормализации данных;
- использовать обратные вызовы `tf.Model.fit()` для обновления веб-интерфейса во время обучения.

2.3.1. Набор данных стоимости жилья в Бостоне

Набор данных стоимости жилья в Бостоне[1] (Boston-housing) представляет собой 500 простых записей с информацией о ценах на недвижимость в Бостоне (штат Массачусетс) и в его окрестностях в конце 1970-х годов. Он уже в течение десятилетий используется как стандартный набор данных для знакомства с задачами статистики и машинного обучения. Каждая запись в этом наборе содержит несколько числовых характеристик микрорайонов Бостона, включая, например, удаленность местности от ближайшего шоссе, наличие выхода к морю и т. д. В табл. 2.1 приведен точный упорядоченный перечень признаков, включая средние значения каждого из них.

Таблица 2.1. Признаки в наборе данных Boston-housing

Индекс	Краткое название признака	Описание признака	Среднее значение	Диапазон (мин — макс)
0	CRIM	Показатель преступности	3,62	88,9
1	ZN	Доля земли под жилыми постройками, разбитой на участки более 2300 м^2	11,4	100
2	INDUS	Доля акров, занимаемых нерозничным бизнесом (промышленностью) в городе	11,2	27,3

[1] *Harrison D., Rubinfeld D.* Hedonic Housing Prices and the Demand for Clean Air // Journal of Environmental Economics and Management. Vol. 5. 1978. Pp. 81–102. http://mng.bz/1wvX.

Индекс	Краткое название признака	Описание признака	Среднее значение	Диапазон (мин — макс)
3	CHAS	Прилегает ли местность к реке Чарльз	0,0694	1
4	NOX	Концентрация окиси азота в воздухе (в десятимиллионных долях)	0,555	0,49
5	RM	Среднее число комнат в жилых домах	6,28	5,2
6	AGE	Доля неарендуемых жилищ, построенных до 1940 года	68,6	97,1
7	DIS	Взвешенные расстояния до пяти центров концентрации рабочих мест Бостона	3,80	11,0
8	RAD	Показатель удобства доступа к радиальным дорогам	9,55	23,0
9	TAX	Ставка налога на $10 000	408	524,0
10	PTRATIO	Число учащихся на одного преподавателя	18,5	9,40
11	LSTAT	Процент работающих мужчин без среднего школьного образования	12,7	36,2
12	MEDV	Медианная стоимость неарендуемых домов в тысячах долларов	22,5	45

В этом разделе мы создадим и обучим систему машинного обучения для оценки медианной стоимости домов в микрорайонах Бостона (MEDV) по заданным входным признакам, а также выясним качество ее работы. Можете считать ее системой оценки стоимости недвижимости на основе измеримых показателей микрорайонов.

2.3.2. Получаем с GitHub и запускаем проект Boston-housing

Поскольку эта задача несколько сложнее примера с предсказанием времени скачивания и включает больше составных элементов, мы начнем с того, что предоставим вам ее решение в виде репозитория с работающим кодом, а затем пройдемся по нему. Если вы уже хорошо ориентируетесь в том, как работает система управления исходным кодом GitHub и система управления пакетами npm/Yarn, то можете пропустить этот подраздел. Дополнительная информация об основной структуре проектов JavaScript приведена в инфобоксе 2.3.

Начнем с получения копий HTML-, JavaScript-файлов и файлов конфигурации, клонировав репозиторий проекта из GitHub[1]. За исключением простейших (размещенных на CodePen), все примеры в книге собраны и разбиты по каталогам в одном из двух репозиториев Git: `tensorflow/tfjs-examples` и `tensorflow/tfjs-models`. Для клонирования необходимого для этого примера репозитория на локальную машину и смены рабочего каталога на каталог проекта boston-housing выполните следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/boston-housing
```

ИНФОБОКС 2.3. Базовая структура проектов JavaScript для примеров из книги

Стандартная структура проекта для наших примеров включает три важных типа файлов. Первый — HTML. Используемые HTML-файлы, сокращенные до абсолютного минимума, будут служить главным образом каркасами для всего нескольких компонентов. Чаще всего в проекте будет только один HTML-файл — `index.html`, — включающий несколько тегов `<div>`, возможно, несколько элементов UI и тег `source` для подтягивания в него JavaScript-кода, например, из файла `index.js`.

JavaScript-код будет разбит на несколько файлов ради удобства чтения и хорошего стиля. В случае проекта Boston-housing код, отвечающий за обновление визуальных элементов, располагается в `ui.js`, а код скачивания данных — в `data.js`. На оба файла мы ссылаемся из файла `index.js` с помощью операторов `import`.

Третий важный тип файлов, с которыми нам предстоит работать, — файл пакета метаданных `.json`. Он нужен системе управления пакетами npm (http://www.npmjs.com/). Если вам еще не приходилось работать с npm или Yarn, рекомендуем просмотреть документацию npm для начинающих по адресу https://docs.npmjs.com/about-npm и разобраться в ней настолько, чтобы уметь собирать и запускать примеры кода. В качестве системы управления пакетами мы будем использовать Yarn (https://yarnpkg.com/en/), но если вам больше нравится npm, то можете работать с ним.

Обратите внимание на следующие важные файлы внутри репозитория:

- `index.html` — корневой HTML-файл, содержащий корневой узел DOM и обеспечивающий вызов сценариев JavaScript;
- `index.js` — корневой JavaScript-файл, отвечающий за загрузку данных, описание модели и цикла обучения, в котором задаются элементы UI;
- `data.js` — реализация структур данных, необходимых для скачивания набора Boston-housing и обращения к нему;

[1] Исходный код примеров из этой книги открыт и размещен на сайтах github.com и codepen.io. Если вам нужна дополнительная информация о том, как использовать инструментарий управления исходным кодом Git, можете воспользоваться прекрасным руководством, начинающимся на странице https://help.github.com/articles/set-up-git. Если вы обнаружили ошибку или хотели бы помочь нам в прояснении каких-либо моментов, пожалуйста, присылайте исправления через запросы на внесение изменений GitHub.

- `ui.js` — реализация точек подключения UI для связывания элементов UI с выполняемыми действиями; описание настроек графика;
- `normalization.js` — численные процедуры (например, для вычитания из данных среднего значения);
- `package.json` — стандартное описание пакета npm, содержащее информацию о зависимостях, необходимых для сборки и запуска примера (в частности, TensorFlow.js!).

Обратите внимание, что мы не следуем стандартной практике размещения HTML- и JavaScript-файлов в отдельные каталоги в соответствии с их типом. Хотя подобная практика и рекомендуется для более крупных репозиториев, для маленьких примеров из этой книги и репозитория http://github.com/tensorflow/tfjs-examples она скорее все усложняет.

Для запуска примера воспользуйтесь Yarn:

```
yarn && yarn watch
```

В результате должна открыться новая вкладка браузера, указывающая на один из портов `localhost`, где будет выполняться этот пример. Если ваш браузер не отреагирует автоматически, можете самостоятельно перейти по URL, указанному в командной строке. Нажмите кнопку Train Linear Regressor (Обучить линейный регрессор). Это запустит процедуру создания линейной модели и ее обучения на данных набора Boston-housing, а затем выведет анимированный график потерь для обучающего и контрольного наборов данных после каждой эпохи, как показано на рис. 2.11.

В оставшейся части раздела мы рассмотрим основные вехи создания этого примера веб-приложения для бостонских цен на недвижимость. Сначала мы обсудим, как эти данные собираются и обрабатываются для работы с TensorFlow.js. А затем сосредоточим свое внимание на создании, обучении и оценке модели. И наконец, продемонстрируем использование нашей модели для реальных предсказаний на веб-странице.

2.3.3. Доступ к данным о бостонских ценах на недвижимость

В первом проекте (см. листинг 2.1) мы жестко «зашили» данные в виде JavaScript-массивов и преобразовали их в тензоры с помощью функции `tf.tensor2d`. Такой вариант подходит для маленького демонстрационного примера, но, разумеется, не масштабируется на большие приложения. Обычно JavaScript-разработчики имеют дело с данными, размещаемыми в каком-либо сериализованном формате по какому-то URL (возможно, локальному). Например, набор данных о бостонских ценах на недвижимость доступен всем бесплатно в формате CSV в Google Cloud по следующим URL:

- https://storage.googleapis.com/tfjs-examples/multivariate-linear-regression/data/train-data.csv;
- https://storage.googleapis.com/tfjs-examples/multivariate-linear-regression/data/train-target.csv;
- https://storage.googleapis.com/tfjs-examples/multivariate-linear-regression/data/test-data.csv;
- https://storage.googleapis.com/tfjs-examples/multivariate-linear-regression/data/test-target.csv.

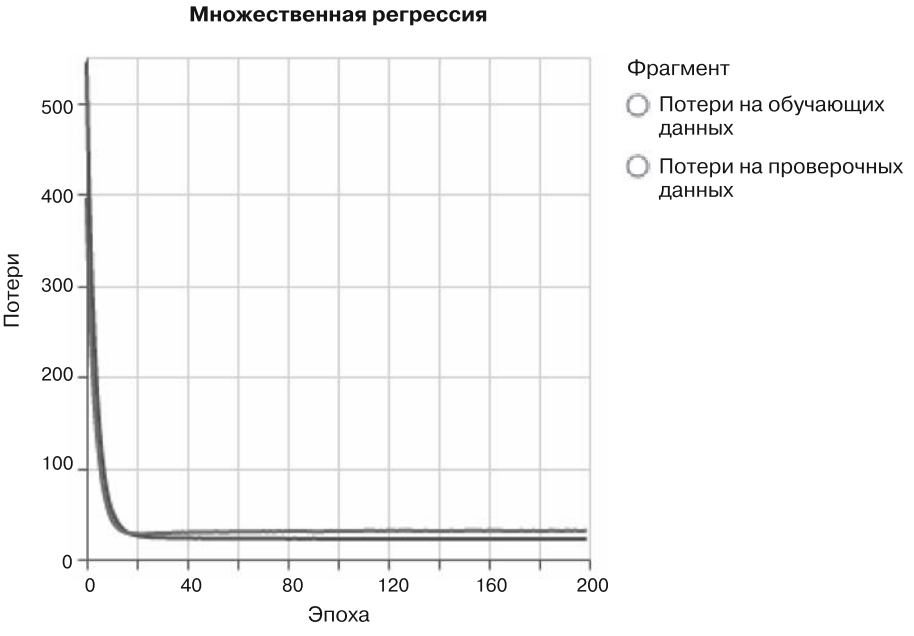


Рис. 2.11. Пример линейной регрессии для бостонских цен на недвижимость из tfjs-examples

Все данные заранее разбиты случайным образом на обучающий и контрольный фрагменты. Примерно две трети примеров данных включены в обучающий фрагмент, а оставшаяся треть отводится на независимую оценку обученной модели. Кроме того, для каждого из фрагментов целевой признак вынесен в особый CSV-файл, отдельно от остальных признаков, в результате чего получились четыре названия файлов, перечисленные в табл. 2.2.

Таблица 2.2. Названия файлов в соответствии с фрагментом и содержимым для набора данных о бостонских ценах на недвижимость

		Признаки (12 чисел)	**Цель (1 число)**
Разбиение на тестовые и контрольные данные	**Обучение**	train-data.csv	train-target.csv
	Проверка	test-data.csv	test-target.csv

Чтобы подтянуть эти данные в наше приложение, необходимо их скачать и преобразовать в тензор соответствующего типа и формы. В файле `data.js` проекта Boston-housing для этой цели описан класс `BostonHousingDataset`. Он абстрагирует потоковые операции над набором данных и предоставляет API для извлечения неформатированных данных в виде числовых матриц. Внутри класса используется

общедоступная библиотека с открытым исходным кодом PapaParse (см. http://www.papaparse.com/) для потоковой передачи и разбора удаленных CSV-файлов. После загрузки и разбора файла библиотека возвращает массив массивов чисел, который затем преобразуется в тензор с помощью того же API, что и в первом примере, как показано в листинге 2.7 — слегка сокращенной выдержке из файла `index.js` с упором на интересующие нас места.

Листинг 2.7. Преобразование данных проекта Boston-housing в тензоры в файле index.js

```
// Инициализация объекта BostonHousingDataset, описанного в файле data.js
const bostonData = new BostonHousingDataset();
const tensors = {};

// Преобразование загруженных CSV-данных типа number[][] в двумерные тензоры
export const arraysToTensors = () => {
  tensors.rawTrainFeatures = tf.tensor2d(bostonData.trainFeatures);
  tensors.trainTarget = tf.tensor2d(bostonData.trainTarget);
  tensors.rawTestFeatures = tf.tensor2d(bostonData.testFeatures);
  tensors.testTarget = tf.tensor2d(bostonData.testTarget);
}

// Запуск асинхронной загрузки данных после загрузки страницы
const tensors = {};
document.addEventListener('DOMContentLoaded', async () => {
  await bostonData.loadData();
  arraysToTensors();
}, false);
```

2.3.4. Точная формулировка задачи проекта Boston-housing

Теперь, когда у нас есть доступ к данным в нужном виде, не помешает более точно сформулировать задачу. Мы говорили, что хотели бы предсказать MEDV на основе прочих полей, но как нам определить, насколько хорошо мы справились с этой задачей? Как отличить хорошую модель от еще лучшей?

Используемая в первом примере метрика `meanAbsoluteError` одинаково учитывает все ошибки. Если набор состоял всего из десяти примеров данных и мы сделали предсказания для всех десяти, причем в девяти случаях правильно, а в десятом ошиблись на 30, то `meanAbsoluteError` будет равна 3 (поскольку 30 / 10 = 3). Если же наши предсказания отклонялись от истинного значения на 3 для каждого примера данных, то `meanAbsoluteError` тоже будет равна 3. Подобный принцип «равенства ошибок» может показаться единственным однозначно правильным вариантом, но есть немало причин выбрать другие метрики потерь, а не `meanAbsoluteError`.

Еще один вариант: придавать крупным погрешностям больший вес, чем маленьким. Вместо среднего значения абсолютной погрешности мы можем использовать среднее значение *квадратичной* погрешности.

Возвращаясь к сценарию с десятью примерами данных, видим, что при подходе со средней квадратичной погрешностью (MSE) потери будут меньше в случае равной 3

погрешности на каждом примере данных ($10 \times 3^2 = 90$), чем в случае ошибки, равной 30, на одном-единственном примере ($1 \times 30^2 = 900$). Из-за чувствительности к крупным ошибкам квадратичная погрешность более чувствительна к аномальным значениям примеров данных, чем абсолютная. Оптимизатор, подгоняющий модель для минимизации MSE, будет предпочитать модели с систематическими малыми ошибками, а не модели, которые изредка выдают очень плохие оценки. Разумеется, для обеих мер погрешности лучшими считаются модели, которые вообще не дают ошибок! Однако если приложение, возможно, очень чувствительно к сильно неточным аномальным значениям, то лучше выбрать MSE, а не MAE. Есть и другие, чисто технические причины, почему лучше может оказаться MSE или MAE, но для нас они сейчас неважны. В этом примере мы для разнообразия воспользуемся MSE, но MAE тоже подошла бы.

Прежде чем продолжить, нам необходимо вычислить эталонную оценку потерь. Если мы не знаем погрешность для очень простой оценки, то не сможем и вычислить ее для более сложной модели. В качестве нашего «лучшего наивного предсказания» воспользуемся средней ценой недвижимости и вычислим погрешность для сценария, когда это значение всегда предсказывается.

Листинг 2.8. Вычисление эталонных потерь при использовании в качестве предсказания средней цены

```
export const computeBaseline = () => {
  const avgPrice = tf.mean(tensors.trainTarget);
  console.log(`Average price: ${avgPrice.dataSync()[0]}`);

  const baseline =
      tf.mean(tf.pow(tf.sub(
          tensors.testTarget, avgPrice), 2));
  console.log(
      `Baseline loss: ${baseline.dataSync()[0]}`);
};
```

Вычисляем среднюю стоимость

Вычисляем среднеквадратичную погрешность на контрольных данных. Вызовы sub(), pow и mean() — отдельные шаги вычисления такой погрешности

Выводим значение эталонных потерь

Поскольку TensorFlow.js оптимизирует вычисления за счет выполнения их на GPU, тензоры не всегда будут доступны CPU. Вызов `dataSync` в листинге 2.8 указывает библиотеке TensorFlow.js завершить вычисления тензора и извлечь значение из GPU в CPU, чтобы можно было его вывести или как-либо еще совместно использовать с операцией, не имеющей отношения к TensorFlow.

В результате выполнения кода из листинга 2.8 в консоль выводится следующее:

```
Average price: 22.768770217895508
Baseline loss: 85.58282470703125
```

Получается, что частота ошибок при «наивном» предсказании составляет примерно 85,58. У модели, которая всегда выдавала бы 22,77, MSE составляла бы 85,58 на контрольных данных. Опять же обратите внимание, что мы вычисляем метрику на обучающих данных и оцениваем ее на контрольных данных во избежание необъективной систематической ошибки.

Средняя *квадратичная* погрешность составляет 85,58, так что для получения средней погрешности необходимо извлечь из нее квадратный корень. Квадрат-

ный корень от 85,58 равен примерно 9,25. Следовательно, можно утверждать, что наша (константная) оценка будет смещена (вниз или вверх) примерно на 9,25. А поскольку суммы здесь указываются в тысячах долларов США (согласно табл. 2.1), то при константном прогнозе мы ошибемся примерно на $9250. Если подобная погрешность для нашего приложения допустима, можно на этом и остановиться! Грамотный специалист-практик по машинному обучению знает, когда лучше избежать ненужных усложнений. Но пусть наше приложение — оценщик стоимости недвижимости должно выдавать более близкие оценки. Попробуем подогнать линейную модель к нашим данным и посмотрим, получится ли у нас MSE лучше 85,58.

2.3.5. Небольшое отступление: нормализация данных

В данных о бостонской недвижимости можно видеть широкий разброс значений признаков. Значения NOX находятся в диапазоне от 0,4 до 0,9, а TAX — от 180 до 711. При подгонке линейной регрессии оптимизатор пытается найти такой весовой коэффициент для каждого признака, чтобы сумма признаков, умноженных на веса, примерно равнялась стоимости недвижимости. Напомним, что в поиске этих весовых коэффициентов оптимизатор следует за градиентом в пространстве весов. И если масштаб отдельных признаков сильно отличается от масштаба прочих, то некоторые веса окажутся намного «чувствительнее» других. Очень малое перемещение в одном направлении будет менять выходной сигнал намного сильнее очень большого перемещения в другом. Это может привести к неустойчивости и сильно затруднить подгонку модели.

В качестве контрмеры необходимо сначала *нормализовать* (normalize) данные. То есть масштабировать признаки до нулевого среднего значения и единичного среднеквадратичного отклонения. Подобная разновидность нормализации встречается очень часто, ее иногда называют *стандартизованным преобразованием* (standard transformation) или *нормализацией по z-оценке* (z-score normalization). Алгоритм очень прост — сначала вычисляется среднее значение каждого признака, вычитается из его исходного значения, в результате чего среднее значение признака становится равно 0. Далее вычисляется отношение признака (с уже вычтенным средним значением) к его среднеквадратичному отклонению. В псевдокоде это выглядит так:

```
нормализованный_признак = (признак – среднее_значение(признак)) /
среднеквадратичное_отклонение(признак)
```

Например, нормализованная версия признаков `[10, 20, 30, 40]` будет равна приблизительно `[-1.3, -0.4, 0.4, 1.3]`, явно с нулевым средним значением; и на глаз среднеквадратичное отклонение также около 1. В примере с ценами на бостонскую недвижимость код нормализации вынесен в отдельный файл `normalization.js`, содержимое которого приведено в листинге 2.9. В нем приводятся две функции, одна для вычисления среднего значения и среднеквадратичного отклонения на основе передаваемого тензора второго ранга, а вторая — для нормализации тензора на основе переданных в нее среднего значения и среднеквадратичного отклонения.

Листинг 2.9. Нормализация данных: нулевое среднее значение, единичное среднеквадратичное отклонение

```
/**
 * Вычисляет среднее значение и среднеквадратичное отклонение
 * каждого столбца массива.
 *
 * @param {Tensor2d} data Набор данных для вычисления среднего значения
 *                        и среднеквадратичного отклонения каждого из столбцов
 *                        по отдельности.
 *
 * @returns {Object} Содержит среднее значение и среднеквадратичное отклонение
 *                   каждого из векторов в виде одномерных тензоров.
 */
export function determineMeanAndStddev(data) {
  const dataMean = data.mean(0);
  const diffFromMean = data.sub(dataMean);
  const squaredDiffFromMean = diffFromMean.square();
  const variance = squaredDiffFromMean.mean(0);
  const std = variance.sqrt();
  return {mean, std};
}

/**
 * Нормализует набор данных на основе заданных среднего значения
 * и среднеквадратичного отклонения путем вычитания среднего значения
 * и деления на среднеквадратичное отклонение.
 *
 * @param {Tensor2d} data: Нормализуемые данные.
 *    Форма: [numSamples, numFeatures]
 * @param {Tensor1d} mean: Ожидаемое среднее значение данных. Форма
[numFeatures]
 * @param {Tensor1d} std: Ожидаемое среднеквадратичное отклонение.
 * Форма [numFeatures]
 *
 * @returns {Tensor2d}: Тензор такой же формы, как у данных,
 * но каждый из столбцов нормализован к нулевому среднему значению
 * и единичному среднеквадратичному отклонению.
 */
export function normalizeTensor(data, dataMean, dataStd) {
  return data.sub(dataMean).div(dataStd);
}
```

Рассмотрим внимательнее эти функции. Функция `determineMeanAndStddev` принимает на входе тензор `data` ранга 2. По традиции первое его измерение соответствует *примерам данных* — каждый индекс соответствует отдельному, независимому примеру данных. Второе измерение — измерение *признаков*: его 12 элементов соответствуют 12 входным признакам (CRIM, ZN, INDUS и т. д.). Чтобы вычислить среднее значение каждого признака по отдельности, мы вызываем:

```
const dataMean = data.mean(0);
```

Ноль в этом вызове означает, что среднее значение вычисляется по первому измерению (измерению с индексом 0). Напомним, что `data` — тензор ранга 2. Первая координата, координата батча, представляет собой измерение примеров данных.

Переходя вдоль этой оси координат от первого элемента ко второму, а затем и к третьему, мы ссылаемся на различные примеры данных, в нашем случае различные объекты недвижимости. Второе измерение — измерение признаков. При переходе в этом измерении от первого элемента ко второму мы ссылаемся на различные признаки, например CRIM, ZN и INDUS из табл. 2.1. При вычислении среднего значения по оси 0 мы усредняем по измерению примеров данных. И получаем в результате тензор ранга 1, в котором осталась только координата признаков со средними значениями каждого признака. Если же вычислять среднее значение по оси 1, мы тоже получим тензор ранга 1, в котором останется только координата примеров данных, а значения будут соответствовать среднему значению для каждого объекта недвижимости, что для нашего приложения смысла не имеет. Будьте внимательны с координатами, убедитесь, что выполняете вычисления по нужному измерению, — это очень распространенный источник ошибок.

Если у нас в этом месте создана точка останова[1], можно воспользоваться консолью JavaScript, чтобы посмотреть вычисленные средние значения. Как видим, средние значения очень близки к средним значениям для набора данных в целом. Это значит, что наша обучающая выборка была вполне репрезентативна:

```
> dataMean.shape
[12]
> dataMean.print();
    [3.3603415, 10.6891899, 11.2934837, 0.0600601, 0.5571442, 6.2656188,
    68.2264328, 3.7099338, 9.6336336, 409.2792969, 18.4480476, 12.5154343]
```

В следующей строке из данных вычитается среднее значение (с помощью `tf.sub`), чтобы мы могли получить их центрированную версию:

```
const diffFromMean = data.sub(dataMean);
```

Если вы недостаточно внимательно следили за нашим обсуждением, то можете пропустить скрытую в этой строке маленькую хитрость. Видите ли, `data` представляет собой тензор ранга 2 формы `[333, 12]`, а `dataMean` — тензор ранга 1 формы `[12]`. Вообще говоря, вычитать тензоры различной формы нельзя. Впрочем, в данном случае TensorFlow использует транслирование (broadcasting) для расширения формы второго тензора фактически путем его повторения 333 раза и делает именно то, что нужно пользователю, не требуя явного описания. Это очень удобный трюк, но правила совместимости форм для транслирования довольно запутанны. Если вам интересны нюансы транслирования, загляните в инфобокс 2.4.

В следующих нескольких строках функции `determineMeanAndStddev` никаких сюрпризов нет: `tf.square()` возводит все элементы в квадрат, а `tf.sqrt()` извлекает из них квадратный корень. Подробное описание API для этих методов можно найти в справочном руководстве из документации TensorFlow.js: https://js.tensorflow.org/api/latest/. На этой странице также представлены интерактивные виджеты с возможностью редактирования — с их помощью вы можете поэкспериментировать над этими функциями с вашими значениями параметров, как показано на рис. 2.12.

[1] Инструкции по созданию точки останова в Chrome можно найти по адресу http://mng.bz/rPQJ. Найти инструкции для Firefox, Edge и других браузеров вы можете в своей любимой поисковой системе, просто введя «как создать точку останова в...».

Рис. 2.12. Документация по API TensorFlow.js, размещенная по адресу js.tensorflow.org и позволяющая непосредственно здесь изучать в интерактивном режиме возможности API. Благодаря этому можно быстро и просто разобраться, как использовать функции, и изучить запутанные граничные случаи

В этом примере мы написали код так, чтобы сделать его максимально прозрачным для читателя, но функцию `determineMeanAndStddev` можно выразить намного лаконичнее:

```
const std = data.sub(data.mean(0)).square().mean().sqrt();
```

Как видите, TensorFlow дает возможность выполнять множество числовых расчетов без лишнего шаблонного кода.

ИНФОБОКС 2.4. Транслирование

Рассмотрим тензорную операцию, скажем `C = tf.некая_операция(A, B)`, где `A` и `B` — тензоры. По возможности, если нет неоднозначностей, меньший тензор транслируется так, чтобы соответствовать форме большего. Транслирование состоит из двух этапов.

1. В меньший тензор добавляется столько дополнительных осей координат (называемых *осями транслирования*), чтобы его ранг соответствовал рангу большего тензора.
2. Меньший тензор повторяется параллельно этим новым осям координат столько раз, сколько нужно для совпадения его формы с формой большего тензора.

Если говорить о реализации, на самом деле никакого нового тензора не создается — это было бы исключительно неэффективно. Операция повторения тензора является полностью воображаемой — она происходит на алгоритмическом уровне, а не на уровне памяти. Но очень удобно в качестве умозрительной модели представлять себе, что меньший тензор повторяется вдоль новой оси координат.

С помощью транслирования можно в общем случае применять поэлементные операции к двум тензорам, один из которых имеет форму `(a, b, ..., n, n + 1, ... m)`, а второй — форму `(n, n + 1, ... m)`. Транслирование будет автоматически происходить для осей координат с `a` до `n - 1`. Например, в следующем примере поэлементная операция `maximum` путем транслирования применяется к двум случайным тензорам различных форм:

```
x = tf.randomUniform([64, 3, 11, 9]);
y = tf.randomUniform([11, 9]);
z = tf.maximum(x, y);
```

x — случайный тензор формы [64, 3, 11, 9]

y — случайный тензор формы [11, 9]

Форма полученного в результате тензора z — такая же, как у x, — [64, 3, 11, 9]

2.3.6. Линейная регрессия по набору данных Boston-housing

Наши данные нормализованы, и мы провели всю необходимую их обработку для вычисления эталонных потерь. Следующий шаг — создание и подгонка модели в надежде добиться лучших, чем эталонные, результатов. В листинге 2.10 описана модель линейной регрессии, подобная приведенной в разделе 2.1 (файл `index.js`). Код практически такой же, единственное отличие от модели для предсказания времени скачивания — конфигурация `inputShape`, теперь с векторами длиной 12 вместо 1. Для единственного плотного слоя по-прежнему указано `units: 1`, то есть выходной сигнал — одно число.

Листинг 2.10. Описание модели линейной регрессии для проекта Boston-housing

```
export const linearRegressionModel = () => {
  const model = tf.sequential();
  model.add(tf.layers.dense(
      {inputShape: [bostonData.numFeatures], units: 1}));
  return model;
};
```

Напомним, что после описания модели, но перед обучением необходимо указать функцию потерь и оптимизатор, вызвав `model.compile`. Как видно из листинга 2.11, задана функция потерь `'meanSquaredError'`, а у оптимизатора — установленная пользователем скорость обучения. В предыдущем примере параметр оптимизатора был установлен равным строке `'sgd'`, сейчас же он равен `tf.train.sgd(LEARNING_RATE)`. Эта фабричная функция возвращает объект, соответствующий алгоритму оптимизации на основе стохастического градиентного спуска, параметризованный нашим пользовательским значением скорости обучения. Такой паттерн, унаследованный

из Keras, часто применяется в TensorFlow.js. Как вы увидите, он используется для множества настраиваемых опций. Для стандартных, широко известных параметров по умолчанию требуемый объектный тип может заменять строковое значение-индикатор, которое TensorFlow.js заменит нужным объектом с хорошими параметрами по умолчанию. В данном случае строка `'sgd'` будет заменена на `tf.train.sgd(0.01)`. Если нужно дополнительно изменить настройки, пользователь может сформировать объект с помощью фабричной функции и указать в нем требуемые пользовательские значения. Благодаря этому код в большинстве случаев оказывается более лаконичным, но пользователи при необходимости могут переопределять поведение по умолчанию.

Листинг 2.11. Компиляция модели для проекта Boston-housing (из файла index.js)

```
const LEARNING_RATE = 0.01;
model.compile({
    optimizer: tf.train.sgd(LEARNING_RATE),
    loss: 'meanSquaredError'});
```

Теперь можно обучить нашу модель на обучающем наборе данных. В листингах с 2.12 по 2.14 мы воспользуемся дополнительными возможностями вызова метода `model.fit()`, но по существу он делает то же самое, что показано на рис. 2.6. На каждом шаге он выбирает несколько новых примеров данных из входных (`tensors.trainFeatures`) и целевых признаков (`tensors.trainTarget`), вычисляет функцию потерь, после чего обновляет внутренние веса для снижения этих потерь. Процесс повторяется в течение `NUM_EPOCHS` полных проходов по обучающим данным с выбором на каждом шаге `BATCH_SIZE` примеров данных.

Листинг 2.12. Обучение модели на данных о бостонской недвижимости

```
await model.fit(tensors.trainFeatures, tensors.trainTarget, {
    batchSize: BATCH_SIZE
    epochs: NUM_EPOCHS,
});
```

В веб-приложении Boston-housing мы продемонстрировали вам график функции потерь по мере обучения. Для построения этого графика нам понадобился обратный вызов `model.fit()` для обновления UI. API обратных вызовов метода `model.fit()` позволяет пользователю передавать функции обратного вызова, выполняемые в случае конкретных событий. Полный список событий, запускающих обратные вызовы, по состоянию на версию 0.12.0 будет таким: `onTrainBegin`, `onTrainEnd`, `onEpochBegin`, `onEpochEnd`, `onBatchBegin` и `onBatchEnd`.

Листинг 2.13. Обратные вызовы в методе model.fit()

```
let trainLoss;
await model.fit(tensors.trainFeatures, tensors.trainTarget, {
  batchSize: BATCH_SIZE,
  epochs: NUM_EPOCHS,
  callbacks: {
    onEpochEnd: async (epoch, logs) => {
```

```
      await ui.updateStatus(
         `Epoch ${epoch + 1} of ${NUM_EPOCHS} completed.`);
      trainLoss = logs.loss;
      await ui.plotData(epoch, trainLoss);
    }
  }
});
```

Последняя пользовательская настройка, представленная здесь, предназначена для проверочных данных. Проверка[1] — понятие машинного обучения, о котором стоит сказать несколько слов отдельно. В предыдущем примере со временем скачивания мы отделили обучающие данные от контрольных, чтобы получить непредвзятую оценку того, как наша модель будет работать на новых, еще не виденных ею данных. Впрочем, обычно выделяется еще один фрагмент — *проверочные данные* (validation data). Такие данные отделены как от обучающих, так и от контрольных данных. Для чего они используются? Специалист по машинному обучению может воспользоваться результатами модели на проверочных данных и на их основе изменить определенные настройки модели[2], чтобы повысить степень безошибочности на проверочных данных. Все это замечательно, но если цикл выполняется много раз, то модель фактически подстраивается под проверочные данные. И если воспользоваться теми же проверочными данными для оценки итоговой безошибочности модели, результат этой оценки не получится экстраполировать в том смысле, что модель уже видела данные, и результат оценки не обязательно будет адекватно отражать ее работу на новых данных в будущем. Именно поэтому проверочные данные отделяют от контрольных. Основная идея заключается в подгонке модели на обучающих данных и подстройке ее гиперпараметров на базе ее оценки на проверочных данных. А когда результаты будут нас удовлетворять, мы только один раз оценим работу модели на контрольных данных, чтобы получить итоговую, обобщаемую оценку качества ее работы.

Давайте резюмируем, что такое обучающий, проверочный и контрольный наборы данных и как следует использовать их в TensorFlow.js. Не во всех проектах применяются все три эти набора. Зачастую при быстром изучении данных или в исследовательских проектах используются только обучающие и проверочные данные, без выделения набора «чистых» данных для контроля. Такой вариант хотя и менее строгий, но иногда позволяет рациональнее использовать ограниченные ресурсы.

- *Обучающие данные* — для подборки весов модели с помощью градиентного спуска.

  Применение в TensorFlow.js. Обычно обучающие данные передаются с помощью основных аргументов (`x` и `y`) вызова `Model.fit(x, y, config)`.
- *Проверочные данные* — для выбора структуры и гиперпараметров модели.

---

[1] Иногда также называется валидацией. — *Примеч. пер.*

[2] В качестве примеров подобных настроек можно привести количество слоев модели, размер слоев, тип оптимизатора, скорость обучения и т. д. Эти так называемые гиперпараметры модели мы подробнее рассмотрим в подразделе 3.1.2.

Применение в TensorFlow.js. Задавать проверочные данные в `Model.fit` можно двумя способами, в обоих случаях — в виде параметров аргумента `config`. При наличии данных для проверки в явном виде их можно указать в `config.validationData`. А если нужно, чтобы фреймворк выделил часть обучающих данных в качестве проверочных, можно указать, какую часть обучающих данных использовать для этого в `config.validationSplit`. Фреймворк сам позаботится о том, чтобы не задействовать проверочные данные для обучения модели, и никакого пересечения данных не будет.

- *Контрольные данные* — для окончательной, непредвзятой оценки работы модели.

  Применение в TensorFlow.js. Контрольные данные передаются системе через аргументы `x` и `y` вызова `Model.evaluate(x, y, config)`.

В листинге 2.14 наряду с потерями на обучающих данных вычисляются потери на проверочных данных. Поле `validation-Split: 0.2` указывает внутренним механизмам `model.fit()` использовать последние 20 % обучающих данных в качестве проверочных. Для обучения эти данные использоваться не будут (то есть не будут влиять на градиентный спуск).

Листинг 2.14. Включаем проверочные данные в model.fit()

```
let trainLoss;
let valLoss;
await model.fit(tensors.trainFeatures, tensors.trainTarget, {
  batchSize: BATCH_SIZE,
  epochs: NUM_EPOCHS,
  validationSplit: 0.2,
  callbacks: {
    onEpochEnd: async (epoch, logs) => {
      await ui.updateStatus(
          `Epoch ${epoch + 1} of ${NUM_EPOCHS} completed.`);
      trainLoss = logs.loss;
      valLoss = logs.val_loss;
      await ui.plotData(epoch, trainLoss, valLoss);
    }
  }
});
```

Обучение этой модели в течение 200 эпох занимает примерно 11 секунд в браузере на современном ноутбуке. Теперь можно оценить работу модели на нашем контрольном наборе данных и выяснить, превзошла ли она эталонные показатели. В листинге 2.15 показано, как следует использовать `model.evaluate()` для определения эффективности работы модели на наших отложенных контрольных данных, с последующим вызовом наших специально настроенных процедур UI для обновления визуального представления.

Листинг 2.15. Оценка работы модели на контрольных данных и обновление UI (из файла index.js)

```
await ui.updateStatus('Running on test data...');
const result = model.evaluate(
    tensors.testFeatures, tensors.testTarget, {batchSize: BATCH_SIZE});
```

```
const testLoss = result.dataSync()[0];
await ui.updateStatus(
    `Final train-set loss: ${trainLoss.toFixed(4)}\n` +
    `Final validation-set loss: ${valLoss.toFixed(4)}\n` +
    `Test-set loss: ${testLoss.toFixed(4)}`);
```

В этом коде `model.evaluate()` возвращает скалярное значение (то есть тензор ранга 0), содержащее вычисленные на основе контрольного набора данных потери.

В силу случайного характера градиентного спуска ваши результаты могут несколько отличаться, но приведенные ниже результаты вполне характерны:

- итоговые потери на обучающем наборе данных — 21,9864;
- итоговые потери на проверочном наборе данных — 31,1396;
- потери на контрольном наборе данных — 25,3206;
- эталонные потери — 85,58.

Как видно, окончательная, непредвзятая оценка погрешности модели равна примерно 25,3 — намного лучше, чем «наивная» эталонная погрешность 85,6. Напоминаем, что погрешность мы вычисляем с помощью `meanSquaredError`. Извлекаем квадратный корень и видим, что эталонная оценка потерь в среднем демонстрировала погрешность более 9,2, а линейная модель дает ошибку всего 5,0. Весьма существенный прогресс! Если бы доступ к этой информации был во всем мире только у нас, в 1978 году мы были бы самыми успешными торговцами бостонской недвижимостью! Разве что кому-нибудь удалось бы сделать еще более точную оценку...

Если вы дали волю своему любопытству и нажали кнопку Train Neural Network Regressor (Обучить нейросетевой регрессор), то знаете, *насколько* лучшие оценки возможны. В следующей главе мы познакомим вас с нелинейными глубокими моделями и покажем, за счет чего возможны подобные достижения.

2.4. Интерпретация модели

Когда модель уже обучена и способна выполнять обоснованные предсказания, нам становится интересно, чему она научилась. Можно ли как-то заглянуть внутрь модели и узнать, как она понимает данные? Когда модель на основе входных данных предсказывает определенную цену, можно ли найти разумное пояснение того, почему она выдала именно такое значение? В общем случае для больших глубоких сетей понимание модели (называемое также интерпретируемостью модели) остается областью активных исследований, ему посвящено множество стендов и докладов на научных конференциях. Но для подобной простой модели линейной регрессии все просто.

К концу этого раздела вы научитесь:

- извлекать из модели усвоенные веса;
- интерпретировать эти веса и сравнивать их с своим интуитивным представлением о том, какими они *должны* быть.

2.4.1. Выясняем смысл усвоенных весов

Созданная нами в разделе 2.3 простая модель линейной регрессии усвоила 13 параметров, содержащихся в ядре и смещении, аналогично нашей первой линейной модели из подраздела 2.1.3:

```
выходной_сигнал = ядро · признаки + смещение
```

Значения ядра и смещения усваиваются в ходе обучения модели. В отличие от усвоенной в подразделе 2.1.3 *скалярной* линейной функции здесь признаки и ядро — *векторы*, а символ «·» означает *внутреннее произведение* (inner product) — обобщение понятия скалярного умножения на векторы. Внутреннее произведение (называемое часто *скалярным произведением векторов* (dot product)) равно просто сумме произведений соответствующих элементов векторов. Более точное определение внутреннего произведения приведено в псевдокоде в листинге 2.16.

Листинг 2.16. Псевдокод внутреннего произведения

```
function innerProduct(a, b) {
    output = 0;
    for (let i = 0 ; i < a.length ; i++) {
        output += a[i] * b[i];
    }
    return output;
}
```

Из этого следует, что между элементами вектора признаков и элементами ядра существует взаимосвязь. Каждому из элементов признаков, например, «Показатель преступности» или «Концентрация окиси азота», перечисленных в табл. 2.1, в ядре соответствует усвоенное число. Каждое значение рассказывает нам о том, *что* модель узнала про этот признак и как он влияет на выходной сигнал.

Например, если модель усвоила, что `kernel[i]` больше нуля, значит, выходной сигнал увеличивается при росте значения `feature[i]`. И наоборот, если модель усвоила, что `kernel[i]` меньше нуля, значит, чем больше значение `feature[i]`, тем меньше будет предсказываемый выходной сигнал. Очень маленькое усвоенное значение указывает на то, что модель считает влияние соответствующего признака на предсказание ничтожным, а очень большое — что модель придает этому признаку существенный вес и относительно небольшие изменения значения признака сильно повлияют на предсказание[1].

Чтобы конкретизировать эти рассуждения, приведем на рис. 2.13 первые пять, по абсолютному значению, признаков, усвоенных за один проход нашего примера с ценами на бостонскую недвижимость. Во время последующих проходов, вероятно, будут усвоены другие значения вследствие случайности начальных значений. Как видим, значения отрицательны для тех признаков, которые должны по логике от-

[1] Обратите внимание, что подобное сравнение порядков величин возможно лишь в случае нормализации признаков, которую для набора данных Boston-housing мы выполнили.

рицательно сказываться на цене недвижимости, например процент местных жителей, бросающих учебу в школе, и расстояние от недвижимости до места работы. И усвоенные веса положительны для признаков, которые положительно коррелируют с ценой, например для количества комнат в доме.

Процент жителей, бросивших учебу в школе	–3,8119
Расстояние до места работы	–3,7278
Число комнат в доме	2,8451
Расстояние до шоссе	2,2949
Концентрация окиси азота	–2,1190

Рис. 2.13. Пять наиболее крупных по абсолютному значению весов, усвоенных за один проход линейной модели для задачи предсказания цен на бостонскую недвижимость. Обратите внимание на отрицательные значения признаков

2.4.2. Извлекаем из модели внутренние веса

Модульная структура обученной модели сильно упрощает извлечение интересующих нас весов. К ним можно обращаться непосредственно, но для получения исходных значений необходимо пройти несколько слоев API. Важно учесть, что, поскольку значение может обрабатываться в GPU, а обмен информацией между физическими устройствами требует немалых ресурсов, запрос подобных значений производится асинхронно. Выделенный жирным шрифтом код в листинге 2.17 дополняет обратные вызовы `model.fit`, расширяя листинг 2.14 для демонстрации усвоенных весов после каждой эпохи. Мы рассмотрим далее эти вызовы API шаг за шагом.

Листинг 2.17. Обращение к внутренним значениям модели

```
let trainLoss;
let valLoss;
await model.fit(tensors.trainFeatures, tensors.trainTarget, {
  batchSize: BATCH_SIZE,
  epochs: NUM_EPOCHS,
  validationSplit: 0.2,
  callbacks: {
    onEpochEnd: async (epoch, logs) => {
      await ui.updateStatus(
          `Epoch ${epoch + 1} of ${NUM_EPOCHS} completed.`);
      trainLoss = logs.loss;
       valLoss = logs.val_loss;
      await ui.plotData(epoch, trainLoss, valLoss);
      model.layers[0].getWeights()[0].data().then(kernelAsArr => {
```

```
        // console.log(kernelAsArr);
        const weightsList = describeKerenelElements(kernelAsArr);
        ui.updateWeightDescription(weightsList);
      });
    }
  }
});
```

Прежде всего нам нужно получить доступ к правильному слою модели. В данном случае это просто, ведь в нашей модели только один слой, к которому можно обратиться с помощью `model.layers[0]`. Теперь можно получить доступ к внутренним весам с помощью метода `getWeights()`, возвращающего массив весов. В случае плотного слоя результат будет всегда содержать два веса: ядро и смещение, причем именно в таком порядке. Следовательно, получить доступ к нужному тензору можно так:

```
> model.layers[0].getWeights()[0]
```

Теперь можно обратиться к содержимому этого тензора, вызвав его метод `data()`. Вследствие асинхронной природы обмена информацией между GPU и CPU метод `data()` — асинхронный и возвращает промис значения тензора, а не само значение. В листинге 2.17 передаваемый в метод `then()` промиса обратный вызов привязывает значения тензора к переменной `kernelAsArr`. Если раскомментировать вызов `console.log()`, на каждой эпохе в консоль будут однократно выводиться выражения наподобие следующих, содержащие значения ядра:

```
> Float32Array(12) [-0.44015952944755554, 0.8829045295715332,
    0.11802537739276886, 0.9555914402008057, -1.6466193199157715,
    3.386948347091675, -0.36070501804351807, -3.0381457805633545,
    1.4347705841064453, -1.3844640254974365, -1.4223048686981201,
    -3.795234441757202]
```

2.4.3. Нюансы интерпретируемости

Из весов на рис. 2.13 можно сделать определенные выводы. Читающий эту книгу человек, глядя на них, может сказать, что модель усвоила. Например, что признак «Количество комнат в доме» положительно коррелирует с предсказываемой ценой или что признак AGE недвижимости, не перечисленный выше вследствие малого абсолютного значения, менее важен, чем те первые пять признаков. А в силу особенностей человеческого мышления мы часто делаем далеко идущие выводы и воображаем, что эти числа говорят нам больше, чем можно утверждать, основываясь на фактах. Например, подобный анализ может привести к ложным выводам, если два входных признака сильно коррелируют между собой.

Рассмотрим гипотетический пример, в котором один и тот же признак включен во входные данные дважды, возможно, просто по недосмотру. Назовем эти два идентичных признака FEAT1 и FEAT2. Допустим, усвоенные веса для этих признаков равны 10 и –5. Хочется сказать, что увеличение FEAT1 приводит к боль-

шему выходному сигналу, а FEAT2 — наоборот. На самом деле, поскольку эти признаки эквивалентны, если поменять их местами, модель будет выводить точно те же значения.

Еще один нюанс — необходимо всегда различать корреляцию и причинную связь. Представьте себе простую модель предсказания силы дождя по степени влажности крыши. При наличии меры влажности крыши можно, наверное, сделать предсказание относительно осадков за последний час. Однако недостаточно просто налить воды на датчик, чтобы вызвать дождь!

Упражнения

1. Мы выбрали задачу оценки времени с жестко «зашитыми» обучающими данными из раздела 2.1 потому, что данные были приблизительно линейными. Впрочем, в других наборах данных поверхности потерь и их поведение во время обучения могут оказаться другими. Попробуйте заменить данные своими собственными и посмотрите, как среагирует модель. Возможно, вам придется поэкспериментировать со скоростью обучения, начальными значениями и нормализацией, чтобы модель начала сходиться к какому-либо интересному результату.
2. В подразделе 2.3.5 мы потратили немало времени на рассказ о том, почему нормализация так важна и как нормализовать входные данные к нулевому среднему значению и единичной дисперсии. Модифицируйте этот пример, убрав нормализацию, и убедитесь, что модель более не обучается. Можете также модифицировать процедуру нормализации так, чтобы среднее значение данных было ненулевым или среднеквадратичное отклонение было меньше нынешнего, но все же больше единицы. Одни виды нормализации приведут к работоспособной модели, а в результате других модель перестанет сходиться.
3. Хорошо известно, что одни признаки набора данных Boston-housing более полезны для предсказания целевого признака, чем другие. Некоторые из его признаков представляют собой просто шум в том смысле, что не несут информации, полезной для предсказания цен на недвижимость. Если бы нам пришлось оставить из всех признаков лишь один, какой следует выбрать? А если оставить два, то какие? Поэкспериментируйте с кодом из нашего примера и выясните это.
4. Расскажите, почему оптимизация модели путем обновления весов с помощью градиентного спуска лучше, чем при случайном выборе параметров.
5. Пример Boston-housing выводит пять наиболее крупных по абсолютному значению весов. Попробуйте модифицировать код так, чтобы выводить признаки, соответствующие самым маленьким весам. Как вы думаете, почему они малы? Если вам зададут вопрос о том, почему эти весовые коэффициенты так малы, что вы ответите? И о каких нюансах интерпретации значений предостережете собеседника?

Резюме

- С помощью TensorFlow.js можно легко, пятью строками JavaScript-кода, создать, обучить и оценить простую модель ML.
- Градиентный спуск, алгоритмический фундамент глубокого обучения, по своей сущности прост и означает лишь многократное обновление параметров модели маленькими шагами в вычисленном направлении, что максимально улучшает подгонку модели.
- Поверхность потерь модели иллюстрирует, насколько хорошо модель подогнана, для целой сетки значений параметров. Рассчитать поверхность потерь не всегда возможно из-за высокой размерности пространства параметров, но она очень наглядна и хорошо демонстрирует работу машинного обучения.
- Одного плотного слоя вполне достаточно для решения некоторых простых задач и получения неплохих результатов на реальных задачах, связанных с ценами на недвижимость.

3 Вводим нелинейность: теперь не только взвешенные суммы

В этой главе

- Что такое нелинейность и почему введение нелинейности в скрытые слои нейронной сети расширяет ее возможности и повышает точность предсказания.
- Что такое гиперпараметры и как их подбирать.
- Бинарная классификация с нелинейностью в выходном слое на примере выявления фишинговых сайтов.
- Многоклассовая классификация и ее отличия от бинарной классификации на примере набора данных «Ирисы Фишера».

В этой главе мы продолжим строить нейронные сети, основываясь на информации из главы 2, и научим их усваивать более сложные соответствия признаков меткам. Главным нашим усовершенствованием станет введение *нелинейности* (nonlinearity) — такого соответствия входных данных выходным, которое нельзя описать простой взвешенной суммой входных элементов. Нелинейность серьезно расширяет возможности нейронных сетей по представлению данных и при правильном применении значительно повышает точность предсказаний во многих задачах. Для иллюстрации сказанного воспользуемся набором данных Boston-housing. Кроме того, в этой главе более подробно рассматриваются вопросы *пере-* и *недообучения*, благодаря чему вы узнаете, как обучить модели, которые не только будут демонстрировать хорошие результаты на обучающих данных, но и давать высокую точность на данных, не встречавшихся им во время обучения, что и является в конечном счете главным показателем качества работы модели.

3.1. Нелинейность: что это такое и где может пригодиться

Давайте продолжим с того момента, на котором остановились в предыдущей главе при работе с примером набора данных Boston-housing. Как вы видели, обученные модели с одним плотным слоем дают MSE, соответствующие ошибке в оценке, равной примерно $5000. Можно ли улучшить этот результат? Безусловно. Чтобы усовершенствовать модель для набора данных Boston-housing, мы добавим в нее еще один плотный слой, как показано в листинге 3.1 (из файла `index.js` примера Boston-housing).

Листинг 3.1. Описание двухслойной нейронной сети для задачи предсказания цен на недвижимость в Бостоне

```
export function multiLayerPerceptronRegressionModel1Hidden() {
  const model = tf.sequential();
  model.add(tf.layers.dense({
    inputShape: [bostonData.numFeatures],
    units: 50,
    activation: 'sigmoid',
    kernelInitializer: 'leCunNormal'
  }));
  model.add(tf.layers.dense({units: 1}));

  model.summary();
  return model;
};
```

Определяет способ задания начальных значений ядра; см. обсуждение его выбора путем оптимизации гиперпараметров в подразделе 3.1.2

Добавляем скрытый слой

Выводим текстовую сводку топологии модели

Чтобы посмотреть на эту модель в действии, сначала выполните команду `yarn && yarn watch`, как упоминалось в главе 2. Когда веб-страница откроется, нажмите кнопку Train Neural Network Regressor (1 Hidden Layer) (Обучить нейросетевой регрессор (1 скрытый слой)) в UI для запуска обучения модели.

Эта модель представляет собой двухслойную нейронную сеть. Первый слой — плотный, из 50 нейронов, с заданной пользователем функцией активации и начальным значением для ядра, которые мы обсудим в подразделе 3.1.2. Этот слой — *скрытый* в том смысле, что его выходной сигнал не виден непосредственно извне модели. Второй слой — плотный, с функцией активации по умолчанию (линейная функция активации) и структурно представляет собой тот же самый слой, что мы использовали в чисто линейной модели в главе 2. Он представляет собой *выходной слой*, поскольку его выходной сигнал является и окончательным выходным сигналом модели, возвращаемым ее методом `predict()`. Наверное, вы обратили внимание, что в названии функции в коде наша модель называется *многослойным перцептроном* (multilayer perceptron, MLP). Это распространенный термин для нейронных сетей с: 1) простой топологией без циклов (так называемые *сети прямого распространения* (feedforward networks)) и 2) по крайней мере одним скрытым слоем. Все модели, которые вы встретите в этой главе, подпадают под это определение.

Новым для нас в листинге 3.1 является вызов `model.summary()`. Это средство диагностики/визуализации, предназначенное для вывода в консоль топологии моделей

TensorFlow.js (либо на инструментальной панели разработчика в браузере, либо в стандартном потоке вывода в Node.js). Вот что было сгенерировано для нашей двухслойной модели:

```
Layer (type)                 Output shape              Param #
=================================================================
dense_Dense1 (Dense)         [null,50]                 650
_________________________________________________________________
dense_Dense2 (Dense)         [null,1]                  51
=================================================================
Total params: 701
Trainable params: 701
Non-trainable params: 0
```

Основная информация в этой сводке.

- Названия и типы слоев (первый столбец).
- Форма выходного сигнала для каждого слоя (второй столбец). Первое измерение (измерение батча) в этих формах практически всегда `null`, символизируя тем самым неопределенный/переменный размер батча.
- Количество весовых параметров для каждого слоя (третий столбец). Здесь указывается количество всех отдельных чисел, составляющих весовые коэффициенты данного слоя. Для слоев с более чем одним весовым коэффициентом оно равно сумме по всем весам. Например, первый плотный слой в данном примере содержит два весовых коэффициента: ядро формы `[12, 50]` и смещение формы `[50]`, так что в результате получается `12 * 50 + 50 = 650` параметров.
- Общее число весовых параметров модели (внизу сводки), а далее — его распределение на обучаемые и необучаемые параметры. Приведенные ранее примеры включали только обучаемые параметры, относящиеся к обновляемым при вызове `tf.Model.fit()` весам модели. Мы обсудим необучаемые веса, когда будем говорить о переносе обучения и точной подстройке модели в главе 5.

Для чисто линейной модели из главы 2 функция `model.summary()` выводит следующее. По сравнению с линейной моделью, наша двухслойная модель содержит примерно в 54 раза больше весовых параметров. Большинство дополнительных весов относится к добавленному нами скрытому слою.

```
_________________________________________________________________
Layer (type)                 Output shape              Param #
=================================================================
dense_Dense3 (Dense)         [null,1]                  13
=================================================================
Total params: 13
Trainable params: 13
Non-trainable params: 0
```

Поскольку двухслойная модель включает больше слоев и весовых параметров, чем однослойная, ее обучение и вывод на основе обученной модели требуют больше времени и вычислительных ресурсов. Стоит ли игра свеч? Оправдывает ли повышение точности эти дополнительные затраты? Когда мы обучили эту модель на

протяжении 200 эпох, итоговые MSE на контрольном наборе данных оказались в диапазоне 14–15 (небольшие различия из-за случайности выбора начальных значений), а при использовании линейной модели потери на контрольном наборе составляли примерно 25. Наша новая модель ошибается лишь на $3700–3900, в то время как при чисто линейном подходе погрешность составляла примерно $5000. Ощутимый шаг вперед.

3.1.1. Развиваем чутье на нелинейность в нейронных сетях

Почему точность выросла? Все дело в увеличившейся сложности модели, как демонстрирует рис. 3.1. Во-первых, в ней появился дополнительный слой нейронов — скрытый слой. Во-вторых, этот скрытый слой содержит нелинейную *функцию активации* (задана фрагментом кода `activation: 'sigmoid'`), которой соответствуют квадраты в блоке Б на рис. 3.1. Функция активации[1] представляет собой поэлементное преобразование. Сигма-функция — «сплющивающая» нелинейность, в том смысле, что она «сплющивает» все вещественные значения от $-\infty$ до $+\infty$ в намного меньший диапазон (в данном случае до 0 до +1). Ее математическое уравнение и график приведены на рис. 3.2. Возьмем в качестве примера скрытый плотный слой. Пусть результат матричного умножения и сложения со смещением представляет собой двумерный вектор, состоящий из следующего массива случайных значений:

```
[[1.0], [0.5],… [0.0]]
```

Итоговый выходной сигнал плотного слоя получается путем вызова сигма-функции (`S`) для каждого из 50 элементов по отдельности:

```
[[S(1.0)], [S(0.5)],… [S(0.0)]] = [[0.731], [0.622],… [0.0]]
```

Почему эта функция называется *нелинейной*? Наглядно это можно объяснить тем, что график функции активации не является прямой. Например, график сигма-

[1] Термин «функция активации» родом из биологии и связан с изучением «живых» нейронов, взаимодействующих друг с другом посредством потенциалов действия (action potential, скачков напряжения на мембранах их клеток). Типичный «живой» нейрон получает входные сигналы от нескольких расположенных перед ним нейронов через точки соединения — синапсы (synapses). Расположенные ближе нейроны возбуждают потенциалы действия с различной частотой, в результате чего высвобождаются нейромедиаторы либо открываются/закрываются каналы передачи ионов в синапсах. Это, в свою очередь, приводит к изменению напряжения тока на мембране нейрона-реципиента. Все это до какой-то степени напоминает взвешенные суммы для блоков плотного слоя. Лишь когда потенциал превышает определенное пороговое значение, нейрон-реципиент фактически генерирует потенциалы действия (то есть «активируется») и влияет на состояние расположенных далее нейронов. В этом смысле функция активации настоящего «живого» нейрона напоминает функцию ReLU (см. рис. 3.2, правый блок), состоящую из «мертвой зоны» ниже определенного порогового значения входного сигнала, и растет линейно относительно входного сигнала выше этого порогового значения (по крайней мере до определенного уровня насыщения, не захватываемого функцией ReLU).

функции — кривая (см. рис. 3.2, *левый блок*), а график ReLU склеен из двух прямых сегментов (см. рис. 3.2, *правый блок*). И хотя сигма-функция и ReLU нелинейные, в числе их свойств — гладкость и дифференцируемость во всех точках, что позволяет производить с их помощью обратное распространение ошибки[1]. Без этих свойств невозможно было бы обучать модели, включающие слои с подобной функцией активации.

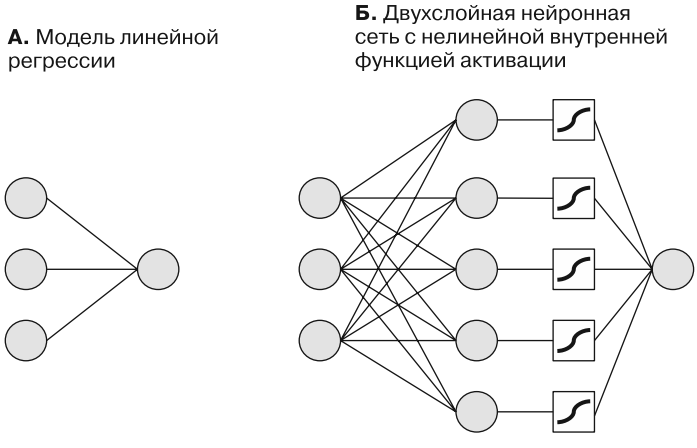


Рис. 3.1. Модель линейной регрессии (блок А) и двухслойная нейронная сеть (блок Б) для набора данных Boston-housing. Ради большей наглядности мы снизили число входных признаков с 12 до 3, а число нейронов скрытого слоя — с 50 до 5 в блоке Б. У каждой модели только один выходной нейрон, поскольку они служат для решения задачи простой (univariate) регрессии (в роли целевой переменной выступает одно число). Блок Б иллюстрирует нелинейную активацию скрытого слоя модели

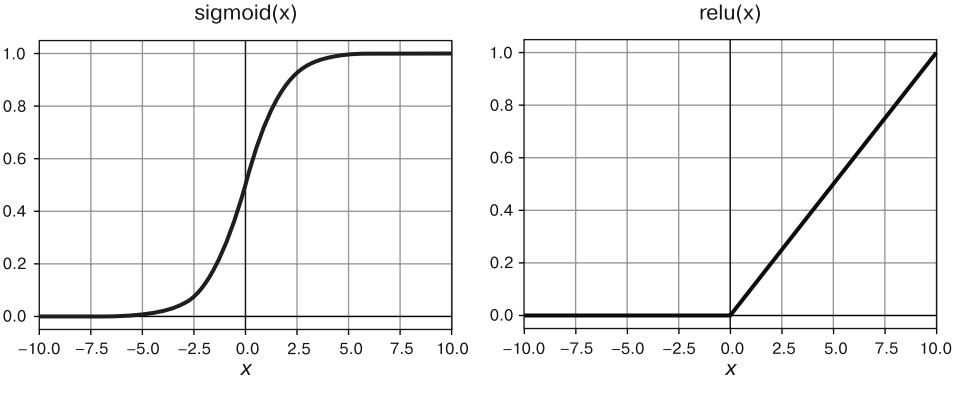


Рис. 3.2. Две функции активации, часто применяемые для глубоких нейронных сетей.
Слева: сигма-функция S(x) = 1 / (1 + e ^ –x).
Справа: функция-выпрямитель (ReLU) relu(x) = {0:x < 0, x:x >= 0}

[1] См. подраздел 2.2.2, чтобы освежить знания о методе обратного распространения ошибки.

Помимо сигма-функции, в глубоком обучении нередко используется еще несколько видов дифференцируемых нелинейных функций. В их числе ReLU и гиперболический тангенс (th). Мы опишем их подробнее, когда встретим в следующих примерах.

Нелинейность и разрешающие возможности модели

Почему нелинейность повышает точность модели? Благодаря нелинейным функциям можно выражать более разнообразные отношения входного и выходного сигналов. Многие из таких отношений на практике оказываются практически линейными, как в нашем примере со временем скачивания из предыдущей главы. Но многие прочие отношения — нет. Привести примеры нелинейных отношений несложно. Рассмотрим отношение между ростом и возрастом человека. Рост меняется с возрастом практически линейно лишь до определенного момента, после которого график изгибается и перестает расти. Или другой вполне возможный сценарий: цены на дома линейно обратно зависят от уровня преступности в микрорайоне, только если уровень преступности находится в определенных пределах. Чисто линейная модель наподобие той, которую мы создали в предыдущей главе, не может точно смоделировать такие отношения, для этой цели гораздо лучше подойдет нелинейная сигма-функция. Конечно, отношение уровня преступности к ценам на дома скорее напоминает перевернутую (убывающую) сигма-функцию, а не исходную, показанную в левом блоке рис. 3.2. Но для нашей нейронной сети не составит никаких трудностей смоделировать это соотношение, поскольку перед сигма-функцией активации и после нее есть линейные функции с подстраиваемыми весами.

Но не утратит ли наша модель способность усваивать линейные отношения из данных при замене линейной функции активации на нелинейную, наподобие сигма-функции? К счастью, ответ — нет. Дело в том, что часть графика сигма-функции (около центра) достаточно близка к прямой линии. Графики других часто используемых нелинейных функций активации, например th и ReLU, также включают линейные или близкие к линейным фрагменты. Если связи между определенными элементами входного и выходного сигналов приближенно линейны, плотный слой с нелинейной функцией активации вполне может усвоить веса и смещения, подходящие для использования почти линейных фрагментов функции активации. Поэтому включение в плотный слой нелинейной функции активации дает чистый выигрыш в смысле диапазона доступных ему для усвоения связей входного и выходного сигналов.

Более того, нелинейные функции можно соединять последовательно и получать, в отличие от линейных, новые нелинейные функции с большими возможностями. *«Последовательно»* здесь означает подачу выходного сигнала одной функции на вход следующей. Пусть дано две линейные функции:

```
f(x) = k1 * x + b1
```

и

```
g(x) = k2 * x + b2
```

Их последовательное соединение (суперпозиция) равносильно описанию новой функции:

```
h(x) = g(f(x)) = k2 * (k1 * x + b1) + b2 = (k2 * k1) * x + (k2 * b1 + b2)
```

Как вы можете видеть, функция `h` осталась линейной, просто с другим ядром (угловым коэффициентом) и другим смещением (точкой пересечения с осью координат), чем у функций `f` и `g`. Угловой коэффициент теперь равен `k2 * k1`, а смещение равно `k2 * b1 + b2`. В результате суперпозиции любого числа линейных функций все равно получается линейная функция.

Взглянем теперь на часто используемую нелинейную функцию активации: ReLU. Внизу рис. 3.3 показано, что произойдет в результате суперпозиции двух нормированных функций ReLU. При суперпозиции двух нормированных функций ReLU получается функция, совершенно непохожая на ReLU. Форма ее графика существенно отличается от формы ReLU. Дальнейшая суперпозиция этой ступенчатой функции с другими функциями ReLU приведет к еще более разнообразному множеству функций, например к оконной функции, функции, состоящей из нескольких окон, функций с окнами наверху более широких окон и т. д. (все это на рис. 3.3 не показано). Путем суперпозиции нелинейных функций наподобие ReLU (одной из чаще всего используемых функций активации) можно создать функции с поразительно широким диапазоном форм. Но при чем здесь нейронные сети? По сути, нейронные сети представляют собой суперпозицию функций. Каждый из слоев нейронной сети можно рассматривать как функцию, а набор слоев — как создание (путем суперпозиции этих функций) более сложной функции — собственно, самой нейронной сети. Из этого должно быть ясно, почему включение в нейронную сеть нелинейных функций активации расширяет диапазон доступных для усвоения моделью отношений входного и выходного сигналов. Становится также интуитивно понятно, зачем нужен часто используемый прием с добавлением дополнительных слоев в нейронную сеть и почему в результате его нередко (но не всегда!) получаются модели, лучше подгоняемые к набору данных.

Диапазон доступных для усвоения моделью машинного обучения отношений входного и выходного сигналов часто называют *разрешающими возможностями* (capacity) модели. Из предыдущего разговора о нелинейности понятно, что разрешающие возможности нейронной сети со скрытыми слоями и нелинейными функциями активации больше, чем у линейного регрессора. Именно поэтому точность нашей двухслойной модели на контрольном наборе данных выше, чем у модели линейной регрессии.

Наверное, вам интересно, можно ли улучшить модель для задачи предсказания цен на бостонскую недвижимость, просто добавив в нейронную сеть больше скрытых слоев, раз уж суперпозиция нелинейных функций активации ведет к повышению разрешающих возможностей (как, например, в нижней части рис. 3.3). Именно это мы и делаем в функции `multiLayerPerceptronRegressionModel2Hidden()` из файла `index.js`. Эта функция закреплена за кнопкой Train Neural Network Regressor (2 Hidden Layers) (Обучить нейросетевой регрессор (2 скрытых слоя)). См. листинг 3.2 с фрагментом кода (из файла `index.js` примера Boston-housing).

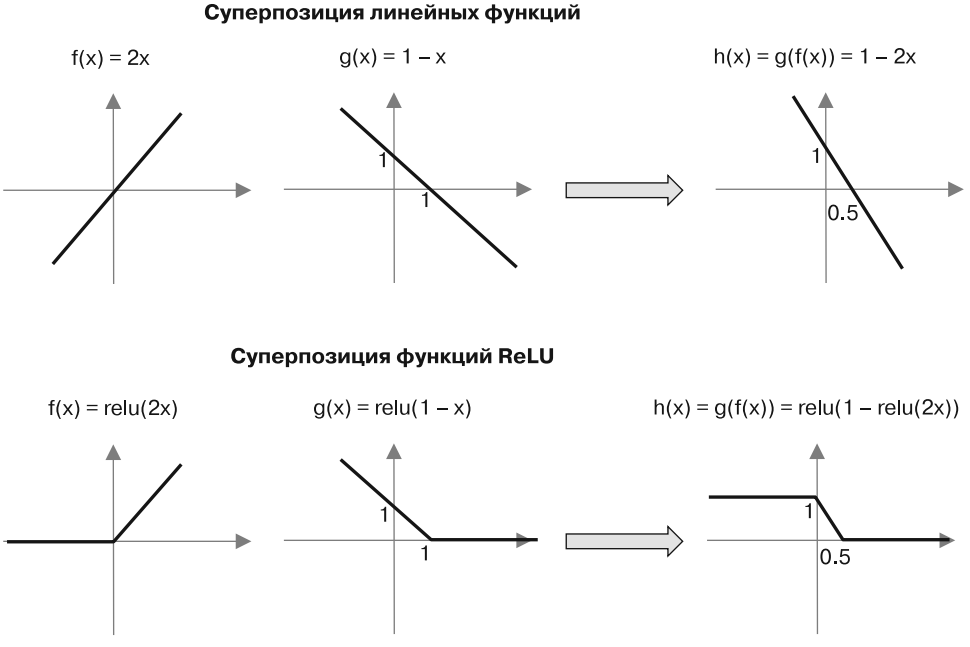


Рис. 3.3. Суперпозиция линейных *(вверху)* и нелинейных функций *(внизу)*. В результате суперпозиции линейных функций всегда получается тоже линейная функция, хотя и с новыми угловыми коэффициентами и точками пересечения с осью координат. В результате же суперпозиции нелинейных функций (таких как ReLU в этом примере) получаются нелинейные функции совершенно другой формы. Из этого рисунка понятно, почему нелинейные функции активации и их суперпозиция в нейронных сетях увеличивает выразительность (то есть разрешающие возможности) последних

Листинг 3.2. Описание трехслойной нейронной сети для задачи предсказания цен на бостонскую недвижимость

```
export function multiLayerPerceptronRegressionModel2Hidden() {
  const model = tf.sequential();
  model.add(tf.layers.dense({
    inputShape: [bostonData.numFeatures],
    units: 50,
    activation: 'sigmoid',
    kernelInitializer: 'leCunNormal'
  }));
  model.add(tf.layers.dense({
    units: 50,
    activation: 'sigmoid',
    kernelInitializer: 'leCunNormal'
  }));
  model.add(tf.layers.dense({units: 1}));

  model.summary();
  return model;
};
```

Добавляем первый скрытый слой

Добавляем еще один скрытый слой

Выводим текстовую сводку топологии модели

Из выведенной функцией `summary()` (здесь мы ее не приводим) информации видно, что модель включает три слоя, то есть на один слой больше, чем модель из листинга 3.1. Кроме того, в ней намного больше параметров: 3251 вместо 701 из двухслойной модели. Дополнительными 2550 весовыми параметрами мы обязаны второму скрытому слою, состоящему из ядра формы `[50, 50]` и смещения формы `[50]`.

Если повторить обучение модели несколько раз, можно понять, в каких рамках находится MSE трехслойных сетей для контрольного набора данных (то есть при оценке качества работы модели): приблизительно 10,8–13,4. Это соответствует погрешности оценки, равной $3280–3660, что лучше, чем у двухслойной модели ($3700–3900). Таким образом, мы еще больше повысили точность предсказания нашей модели, добавив нелинейные скрытые слои, а значит, и расширили ее разрешающие возможности.

Избегаем нагромождения слоев без нелинейностей

Чтобы ощутить важность нелинейной функции активации для нашей усовершенствованной модели Boston-housing, можно убрать ее из модели. Листинг 3.3 отличается от листинга 3.1 только закомментированной строкой, в которой задается сигма-функция активации. В результате удаления пользовательской функции активации в слое применяется линейная функция активации по умолчанию. Все остальные аспекты модели, включая число слоев и весовые параметры, не меняются.

Листинг 3.3. Двухслойная нейронная сеть без нелинейной функции активации

```
export function multiLayerPerceptronRegressionModel1Hidden() {
  const model = tf.sequential();
  model.add(tf.layers.dense({
    inputShape: [bostonData.numFeatures],
    units: 50,
    // activation: 'sigmoid',  <-- Убираем нелинейную функцию активации
    kernelInitializer: 'leCunNormal'
  }));
  model.add(tf.layers.dense({units: 1}));

  model.summary();
  return model;
};
```

Как это изменение повлияло на обучение модели? Как вы увидите, если нажмете снова кнопку Train Neural Network Regressor (1 Hidden Layer) в UI, MSE на контрольном наборе данных взлетает до 25, по сравнению с диапазоном 14–15 при сигма-функции. Другими словами, двухслойная модель без сигма-функции активации демонстрирует такие же результаты, что и однослойный линейный регрессор!

Это подтверждает наши рассуждения по поводу суперпозиции линейных функций. Убрав из первого слоя нелинейную функцию активации, мы получили модель, соответствующую суперпозиции двух линейных функций. Как мы показали выше, результат этой суперпозиции — третья линейная функция, никак не повышающая

разрешающие возможности модели. Следовательно, ничего удивительного, что точность этой модели оказалась такой же, как и у линейной. Отсюда и распространенный подводный камень при создании многослойных нейронных сетей: *не забывайте включать в скрытые слои нелинейные функции активации*. Если этого не сделать, вы только понапрасну потратите вычислительные ресурсы и свое время, а вдобавок рискуете потерять численную устойчивость (видите извилистые кривые потерь в блоке Б на рис. 3.4?). Далее мы увидим, что это относится не только к плотным, но и к другим типам слоев, например сверточным.

Нелинейность и интерпретируемость модели

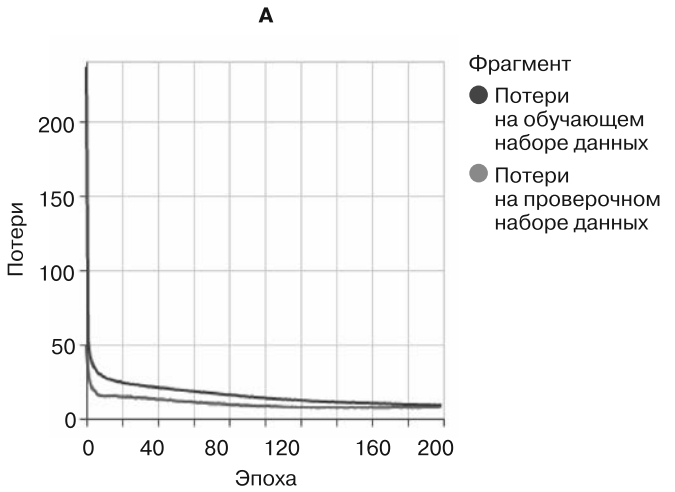
В главе 2 мы показали, что после обучения модели на наборе данных Boston-housing можно изучать ее весовые коэффициенты и истолковывать отдельные параметры достаточно осмысленным образом. Например, значение весового коэффициента, соответствующего признаку «среднее число комнат в жилом доме», — положительное, а веса, соответствующего признаку «уровень преступности», — отрицательное. Знаки подобных весов отражают ожидаемую прямую или обратную зависимости цены дома и соответствующих признаков. А порядок их величин указывает на то, какую значимость придает модель различным признакам. С учетом вышесказанного возникает естественный вопрос: можно ли интуитивно понятным и разумным образом интерпретировать значения весов нелинейной модели, включающей один скрытый слой или более?

API доступа к значениям весовых коэффициентов ничем не отличается для нелинейной и линейной моделей: необходимо просто воспользоваться методом `getWeights()` объекта модели или его компонент — объектов слоев. В случае MLP из листинга 3.1, например, можно просто вставить следующую строку после завершения обучения модели (сразу после вызова `model.fit()`):

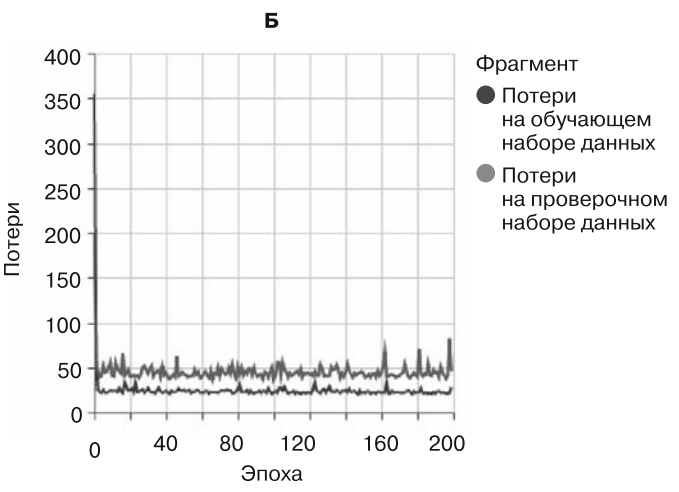
```
model.layers[0].getWeights()[0].print();
```

Эта строка кода выводит в консоль значение ядра первого (скрытого) слоя. Оно представляет собой один из четырех тензоров весов модели, остальные три — смещение скрытого слоя, а также ядро и смещение выходного слоя. Стоит отметить, что его размер больше, чем размер ядра линейной модели, который мы выводили выше:

```
Tensor
    [[-0.5701274, -0.1643915, -0.0009151, ..., 0.313205  , -0.3253246],
     [-0.4400523, -0.0081632, -0.2673715, ..., 0.1735748 , 0.0864024 ],
     [0.6294659 , 0.1240944 , -0.2472516, ..., 0.2181769 , 0.1706504 ],
     [0.9084488 , 0.0130388 , -0.3142847, ..., 0.4063887 , 0.2205501 ],
     [0.431214  , -0.5040522, 0.1784604 , ..., 0.3022115 , -0.1997144],
     [-0.9726604, -0.173905 , 0.8167523 , ..., -0.0406454, -0.4347956],
     [-0.2426955, 0.3274118 , -0.3496988, ..., 0.5623314 , 0.2339328 ],
     [-1.6335299, -1.1270424, 0.618491  , ..., -0.0868887, -0.4149215],
     [-0.1577617, 0.4981289 , -0.1368523, ..., 0.3636355 , -0.0784487],
     [-0.5824679, -0.1883982, -0.4883655, ..., 0.0026836 , -0.0549298],
     [-0.6993552, -0.1317919, -0.4666585, ..., 0.2831602 , -0.2487895],
     [0.0448515 , -0.6925298, 0.4945385 , ..., -0.3133179, -0.0241681]]
```



Итоговые потери на обучающем наборе данных: 9,2346
Итоговые потери на проверочном наборе данных: 8,0922
Потери на контрольном наборе данных: 15,6055
Эталонные потери (meanSquaredError): 85,58



Итоговые потери на обучающем наборе данных: 27,7744
Итоговые потери на проверочном наборе данных: 45,7087
Потери на контрольном наборе данных: 26,0527
Эталонные потери (meanSquaredError): 85,58

Рис. 3.4. Сравнение результатов обучения с сигма-функцией активации (блок А) и без нее (блок Б). Обратите внимание, что исключение сигма-функции активации приводит к большим итоговым потерям на обучающем, проверочном и контрольном наборах данных (на уровне, сравнимом с ранее обсуждавшейся чисто линейной моделью) и менее гладким кривым потерь. Обратите также внимание, что масштабы осей координат в этих двух графиках различаются

Происходит это потому, что скрытый слой состоит из 50 нейронов, в результате чего размер весов составляет `[18, 50]`. Данное ядро включает 900 отдельных весовых параметров, в отличие от `12+1 = 13` параметров ядра линейной модели. Можно ли приписать некий смысл каждому из отдельных весовых параметров? В общем случае — нет. Дело в том, что ни у одного из 50 выходных сигналов скрытого слоя не существует четко определенного смысла. Они представляют собой измерения многомерного пространства, созданного для усвоения (автоматического выявления) в нем моделью нелинейных связей. Человеческий мозг плохо приспособлен для отслеживания нелинейных связей в пространствах подобной размерности. В общем случае весьма непросто описать несколькими словами на доступном непосвященным языке, что делает каждый из нейронов скрытого слоя, или пояснить его вклад в итоговое предсказание глубокой нейронной сети.

Кроме того, учтите, что в приведенной здесь модели был только один скрытый слой. В случае нескольких скрытых слоев, как в модели из листинга 3.2, отношения становятся еще запутаннее, а описать их сложнее. И хотя ведутся исследования по поиску усовершенствованных способов интерпретации смысла скрытых слоев глубоких нейронных сетей[1] и для некоторых классов моделей достигнут значительный прогресс[2], справедливости ради следует сказать, что интерпретация глубоких нейронных сетей — более сложная задача, чем интерпретация неглубоких и некоторых не относящихся к нейронным сетям типов моделей машинного обучения (например, деревьев принятия решений). Выбирая глубокую модель, а не неглубокую, мы, по существу, жертвуем интерпретируемостью в пользу бóльших разрешающих возможностей модели.

3.1.2. Гиперпараметры и их оптимизация

При обсуждении скрытых слоев в листингах 3.1 и 3.2 мы говорили в основном о нелинейной функции активации (сигма-функции). Однако для достижения хороших результатов обучения модели важны и другие параметры конфигурации этого слоя. В их числе количество нейронов (50) и способ инициализации ядра `'leCunNormal'`. Последнее представляет собой особый метод генерации случайных чисел для начальных значений ядра, в зависимости от размера входного сигнала. Он отличается от используемого по умолчанию метода инициализации ядра (`'glorotNormal'`), при котором учитываются размеры как входного, так и выходного сигнала. Возникает естественный вопрос: почему именно этот пользовательский метод инициализации ядра, а не используемый по умолчанию? Почему именно 50 нейронов (а не, скажем, 30)? Эти варианты конфигурации, обеспечивающие оптимальное или близкое

[1] *Ribeiro M. T., Singh S., Guestrin C.* Local Interpretable Model-Agnostic Explanations (LIME): An Introduction. — O'Reilly, 12 Aug. 2016. http://mng.bz/j5vP.

[2] *Olah C. et al.* The Building Blocks of Interpretability // Distill, 6 Mar. 2018, https://distill.pub/2018/building-blocks/.

к таковому качество модели, — результат многократных проб различных сочетаний параметров.

Такие параметры, как количество нейронов, метод задания начальных значений ядра и функция активации, называются *гиперпараметрами* (hyperparameters) модели. Название «гиперпараметры» подчеркивает, что это не весовые параметры модели, обновляемые автоматически во время обучения путем обратного распространения ошибки (то есть вызовов `Model.fit()`). Гиперпараметры модели выбираются один раз и в течение обучения не меняются. Они нередко определяют количество и размер весовых параметров (например, как поле `units` для плотного слоя), начальные значения весовых параметров (как поле `kernelInitializer`) и способ их обновления во время обучения (как переданное в `Model.compile()` поле `optimizer`). Следовательно, это параметры более высокого уровня, чем весовые параметры. Отсюда и название «гиперпараметры».

Помимо размеров слоев и типа весовых параметров, существует множество других типов гиперпараметров модели и ее обучения, например:

- количество плотных слоев в модели наподобие тех, что в листингах 3.1 и 3.2;
- тип используемого для ядра плотного слоя инициализатора;
- использовать ли какую-либо регуляризацию весов (см. раздел 8.1) и если да, то каков коэффициент регуляризации;
- включать ли в модель слои дропаута (см. подраздел 4.3.2) и если да, то какова скорость дропаута;
- тип используемого для обучения оптимизатора (например, `'sgd'` или `'adam'`; см. инфобокс 3.1);
- количество эпох обучения модели;
- скорость обучения оптимизатора;
- снижать ли постепенно скорость обучения оптимизатора по мере обучения и если да, то насколько быстро;
- размер батча для обучения.

Последние пять перечисленных примеров несколько выделяются, поскольку не связаны с самой архитектурой модели как таковой; они представляют собой настройки процесса обучения модели. Тем не менее они влияют на результаты обучения, а значит, могут считаться гиперпараметрами. У моделей, включающих более разнообразные типы слоев (например, сверточные и рекуррентные, обсуждаемые в главах 4, 5 и 9), потенциальное число настраиваемых гиперпараметров еще больше. Отсюда ясно, почему даже у простой модели глубокого обучения количество настраиваемых гиперпараметров может достигать десятков.

Процесс выбора хороших значений гиперпараметров называется *оптимизацией гиперпараметров* (hyperparameter optimization) или *подстройкой* (hyperparameter tuning). Цель оптимизации гиперпараметров — найти набор (гипер)параметров, ведущий к минимальным потерям на проверочном наборе данных после обучения.

К сожалению, пока не существует однозначного алгоритма выбора наилучших гиперпараметров для заданного набора данных и задачи машинного обучения. Сложность в том, что многие гиперпараметры дискретны, а значит, величина потерь на проверочном наборе данных недифференцируема по ним. Например, количество нейронов плотного слоя и количество плотных слоев модели — целые числа; тип оптимизатора — дискретный параметр. Отслеживать во время обучения градиенты даже непрерывных гиперпараметров, относительно которых величина потерь на проверочном наборе данных дифференцируема (например, коэффициентов регуляризации), обычно слишком затратно с вычислительной точки зрения, поэтому производить градиентный спуск в пространстве подобных гиперпараметров на практике обычно не имеет смысла. Оптимизация гиперпараметров остается областью активных исследований, важной для всех практикующих глубокое обучение.

Поскольку стандартной готовой методики/инструмента оптимизации гиперпараметров не существует, специалисты, занимающиеся практическим применением глубокого обучения, используют три подхода. Во-первых, если решаемая задача похожа на другую, хорошо изученную (например, на один из приведенных в данной книге примеров), можно применить к своей задаче эту аналогичную модель, после чего воспользоваться ее гиперпараметрами. А затем достаточно будет производить поиск в относительно небольшом пространстве гиперпараметров возле этой стартовой точки.

Во-вторых, у специалистов с достаточным практическим опытом обычно уже выработано чутье, они способны выдвигать для конкретной задачи обоснованные варианты хороших гиперпараметров модели. И хотя подобный субъективный вариант редко оказывается оптимальным, он служит неплохой отправной точкой для дальнейшего более точного подбора.

В-третьих, в случаях, когда число оптимизируемых гиперпараметров невелико (например, меньше четырех), можно воспользоваться поиском по сетке, то есть перебрать все возможные сочетания гиперпараметров, полностью обучить модель для каждого из них, фиксируя потери на проверочном наборе данных, и выбрать сочетание с минимальными потерями. Например, пусть необходимо найти значения только двух гиперпараметров: 1) количество нейронов в плотном слое и 2) скорость обучения. Можно взять множество значений количества нейронов (`{10, 20, 50, 100, 200}`) и множество скоростей обучения (`{1e-5, 1e-4, 1e-3, 1e-2}`). Произведение этих двух множеств дает `5 * 4 = 20` сочетаний гиперпараметров. Если вы захотите реализовать поиск по сетке самостоятельно, то псевдокод вашей реализации будет выглядеть примерно так, как показано в листинге 3.4.

Листинг 3.4. Псевдокод для простого поиска гиперпараметров по сетке

```
function hyperparameterGridSearch():
  for units of [10, 20, 50, 100, 200]:
    for learningRate of [1e-5, 1e-4, 1e-3, 1e-2]:
       Создать модель, плотный слой которой состоит из `units` нейронов
       Обучить эту модель при оптимизаторе с `learningRate`
```

```
      Вычислить итоговые потери на проверочном наборе validationLoss
      if validationLoss < minValidationLoss
        minValidationLoss := validationLoss
        bestUnits := units
        bestLearningRate := learningRate

  return [bestUnits, bestLearningRate]
```

Как выбрать диапазоны этих гиперпараметров? Это еще один вопрос, на который теория глубокого обучения не дает однозначного ответа. Обычно эти диапазоны выбираются на основе опыта и интуиции конкретного специалиста по глубокому обучению. Кроме того, дополнительные ограничения на них могут накладывать вычислительные ресурсы. Например, модель, включающая плотный слой со слишком большим количеством нейронов, может слишком медленно обучаться или работать в фазе вывода.

Зачастую количество оптимизируемых гиперпараметров настолько велико, что просмотреть все экспоненциально растущее число их сочетаний с вычислительной точки зрения нереально. В подобных случаях понадобятся более изощренные, по сравнению с сеточным поиском, методы, например, случайный поиск[1] и байесовские[2] методы.

3.2. Нелинейность на выходе модели: модели для классификации

Мы привели в качестве примеров две задачи регрессии, в которых предсказывали числовое значение (например, время скачивания или среднюю стоимость дома). Однако есть и другая часто встречающаяся задача машинного обучения: классификация. Часть задач классификации относится к *бинарной классификации* (binary classification), при которой целевой величиной является ответ «да/нет». Мир технологий полон подобных задач. Приведем пару примеров.

- Является ли данное письмо спамом?
- Является ли данная транзакция с кредитной картой допустимой? Или мошеннической?
- Содержит ли заданный аудиофрагмент длительностью секунду конкретное слово?
- Совпадают ли два отпечатка пальца (сняты ли они с одного пальца одного человека)?

---

[1] *Bergstra J., Bengio Y.* Random Search for Hyper-Parameter Optimization // Journal of Machine Learning Research. Vol. 13, 2012. Pp. 281–305. http://mng.bz/WOg1.

[2] *Koehrsen W.* A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning // Towards Data Science. 24 June 2018. http://mng.bz/8zQw.

Другая разновидность задач классификации — *многоклассовая классификация* (multiclass-classification), примеров которой также можно привести немало.

- Относится ли конкретная новостная статья к теме спорта, погоды, игр, политики или к какой-либо другой общей теме?
- Изображена ли на фотографии кошка, собака, лопата и т. д.?
- Определить по данным о движении электронного стилуса, какая буква написана.
- При использовании машинного обучения для простой Atari-подобной компьютерной игры определить, в каком из четырех возможных направлений (вверх, вниз, налево, направо) далее должен двигаться игровой персонаж при текущем положении в игре.

3.2.1. Бинарная классификация

Начнем с простого случая бинарной классификации. Пусть у нас есть данные, на основе которых необходимо принять решение, подразумевающее ответ «да/нет». Для этого примера мы воспользуемся набором данных по фишинговым сайтам (Phishing Website)[1]. Задача состоит в следующем: по набору признаков веб-страницы и ее URL предсказать, не используется ли она для *фишинга* (не имитирует ли другой сайт с целью кражи конфиденциальных данных пользователей).

Набор данных содержит 30 признаков, все — бинарные (представленные значениями –1 или 1) или тернарные (представленные значениями –1, 0 или 1). Мы не станем здесь перечислять все признаки, как делали это для набора данных Boston-housing, а приведем лишь несколько наиболее представительных.

- `HAVING_IP_ADDRESS` — используется ли IP-адрес вместо имени домена (бинарное значение `{-1, 1}`)?
- `SHORTENING_SERVICE` — используется ли сервис сокращенного написания URL (бинарное значение `{-1, 1}`)?
- `SSLFINAL_STATE` — используется ли в URL HTTPS? При этом: 1) издатель сертификата доверенный; 2) издатель сертификата не доверенный; 3) не используется (тернарное значение `{-1, 0, 1}`)?

Набор данных состоит примерно из 5500 обучающих примеров данных и такого же числа контрольных примеров. В обучающем наборе данных почти 45 % примеров данных — позитивные (действительно представляют собой фишинговые веб-страницы). Процент позитивных примеров данных в контрольном наборе данных примерно такой же.

Это фактически простейший из возможных наборов данных — все признаки в наборе данных уже находятся в одинаковом диапазоне, так что не требуется нормализовывать их средние значения и среднеквадратичные отклонения, как мы

[1] *Mohammad R. M., Thabtah F., McCluskey L.* Phishing Websites Features. http://mng.bz/E1KO.

делали для набора данных Boston-housing. Кроме того, количество обучающих примеров данных достаточно велико относительно как числа признаков, так и числа возможных предсказаний (равного двум: да или нет). В целом это неплохая предварительная проверка того, подходит ли набор данных для работы. Если бы не было жалко времени на исследование данных, можно было бы сделать проверку попарной корреляции признаков и выяснить, нет ли избыточной информации; впрочем, нашей модели она не страшна.

Поскольку наши данные схожи с используемыми (после нормализации) данными набора Boston-housing, исходную модель мы также возьмем похожую. Код для этой задачи вы можете найти в каталоге `website-phishing` репозитория tfjs-examples. Для извлечения и запуска примеров можете выполнить следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/website-phishing
yarn && yarn watch
```

Листинг 3.5. Описание модели бинарной классификации для обнаружения фишинга (из файла index.js)

```
const model = tf.sequential();
model.add(tf.layers.dense({
  inputShape: [data.numFeatures],
  units: 100,
  activation: 'sigmoid'
}));
model.add(tf.layers.dense({units: 100, activation: 'sigmoid'}));
model.add(tf.layers.dense({units: 1, activation: 'sigmoid'}));
model.compile({
  optimizer: 'adam',
  loss: 'binaryCrossentropy',
  metrics: ['accuracy']
});
```

Эта модель во многом схожа с многослойной сетью, которую мы создали для задачи предсказания цен на бостонскую недвижимость. Она начинается с двух скрытых слоев (оба — с сигма-функцией активации). Последний (выходной) слой содержит ровно один нейрон, то есть выходной сигнал модели представляет собой одно числовое значение для каждого входного примера данных. Впрочем, главное отличие в том, что в качестве функции активации последнего слоя модели для обнаружения фишинга используется сигма-функция, а не линейная функция активации по умолчанию, как в модели для набора данных Boston-housing. Это значит, что выходной сигнал нашей модели ограничивается числами в диапазоне от 0 до 1, в отличие от модели Boston-housing, которая может выдавать на выходе произвольные числа с плавающей точкой.

Выше мы видели, что сигма-функции активации для скрытых слоев повышают разрешающие возможности модели. Но почему мы воспользовались сигма-функцией активации на выходе этой новой модели? Причина в самой сути задачи бинарной классификации. При бинарной классификации модель обычно должна выдавать

предсказание вероятности позитивной классификации, то есть насколько вероятно, что модель «считает»: данный пример относится к позитивному классу. Как вы помните из школьной математики, вероятность — число в диапазоне между 0 и 1. Есть два преимущества возврата моделью значения ожидаемой вероятности.

- Значение демонстрирует степень уверенности модели в произведенной классификации. Значение сигма-функции 0,5 указывает на полную неопределенность, то есть равную вероятность любой из двух возможных классификаций. Равное 0,6 значение указывает, что, хотя система предсказывает позитивную классификацию, уверенность в ней невелика. Равное же 0,99 значение означает, что модель более чем уверена: этот пример данных относится к позитивному классу и т. д. Таким образом, преобразование выданного моделью значения в окончательный результат не представляет проблем (достаточно пропустить выходной сигнал через сито порогового значения, допустим, 0,5). А теперь представьте себе, насколько сложно было бы найти пороговое значение при очень широком диапазоне изменений выходного сигнала модели.
- Упрощается поиск дифференцируемой функции потерь, которая по заданному выходному сигналу модели и истинным бинарным целевым меткам выдает число — меру погрешности модели. Этот вопрос мы обсудим подробнее, когда будем изучать фактическую бинарную перекрестную энтропию этой модели.

Впрочем, остается вопрос: как втиснуть выходной сигнал нейронной сети в диапазон `[0, 1]`? Последний слой нейронной сети (обычно плотный слой) производит операции матричного умножения (`matMul`) и прибавления смещения (`biasAdd`) своего входного сигнала. У этих операций нет никаких внутренних ограничений, которые бы гарантировали, что результат будет в диапазоне `[0, 1]`. Естественный способ добиться нужного диапазона `[0, 1]` — применить к результату операций `matMul` и `biasAdd` «сплющивающую» нелинейность наподобие сигма-функции.

Еще один новый для нас нюанс кода из листинга 3.5 — тип оптимизатора: `'adam'`, который отличается от использовавшегося в предыдущих примерах оптимизатора `'sgd'`. Чем же `adam` отличается от `sgd`? Как вы помните из подраздела 2.2.2, оптимизатор `sgd` всегда умножает градиенты, полученные путем обратного распространения ошибки, на фиксированное число (скорость обучения, умноженную на –1), чтобы вычислить величины обновлений весовых коэффициентов. У этого подхода есть свои недостатки, включая медленную сходимость к минимуму функции потерь при малой скорости обучения и «зигзагообразные» пути в пространстве весов при определенных особых свойствах (гипер)поверхности потерь. Оптимизатор `adam` как раз и нацелен на устранение этих изъянов `sgd` за счет использования множителя, который хитроумно варьируется в зависимости от истории градиентов (предыдущих итераций обучения). Более того, `adam` обычно обеспечивает лучшую сходимость и меньше зависит от выбранной скорости обучения, по сравнению с `sgd`, для широкого круга типов моделей глубокого обучения, а потому снискал популярность как оптимизатор. В библиотеке TensorFlow.js есть несколько других типов оптимизаторов, часть из которых тоже широко распространены (например, `rmsprop`). Их краткий обзор приведен далее.

ИНФОБОКС 3.1. Поддерживаемые TensorFlow.js оптимизаторы

В следующей таблице приведен краткий список API наиболее используемых типов оптимизаторов в TensorFlow.js, а также простое и интуитивно понятное описание каждого из них.

Распространенные оптимизаторы и их API в TensorFlow.js

Название	API (строковое значение)	API (функция)	Описание
Стохастический градиентный спуск (SGD)	`'sgd'`	`tf.train.sgd`	Простейший оптимизатор, роль множителя для градиентов всегда играет скорость обучения
Метод накопления импульса	`'momentum'`	`tf.train.momentum`	Предыдущие градиенты накапливаются таким образом, что обновление весовых параметров происходит быстрее, если предыдущие градиенты для конкретного параметра выстраиваются в одном направлении, и медленнее, когда они часто меняют направление
RMSProp	`'rmsprop'`	`tf.train.rmsprop`	Для различных весовых параметров модели множитель масштабируется по-разному за счет отслеживания недавней истории изменений среднеквадратического (RMS) значения градиента каждого из весов; отсюда и название
AdaDelta	`'adadelta'`	`tf.train.adadelta`	Скорость обучения для каждого из весовых параметров масштабируется аналогично RMSProp
ADAM	`'adam'`	`tf.train.adam`	Можно рассматривать как сочетание подхода адаптивной скорости обучения и метода накопления импульса
AdaMax	`'adamax'`	`tf.train.adamax`	Аналогичен ADAM, но величины градиентов отслеживаются с помощью несколько иного алгоритма

Возникает очевидный вопрос: какой оптимизатор использовать для конкретной задачи машинного обучения и модели. К сожалению, в сфере глубокого обучения единого мнения по этому поводу пока нет (именно поэтому TensorFlow.js предоставляет все перечисленные в предыдущей таблице оптимизаторы!). На практике имеет смысл начинать с наиболее популярных, включая `adam` и `rmsprop`. При наличии достаточного количества свободного времени и вычислительных ресурсов вы можете считать оптимизатор еще одним гиперпараметром и найти его вариант, обеспечивающий наилучший результат обучения, путем подстройки гиперпараметров (см. подраздел 3.1.2).

3.2.2. Измерение качества работы бинарных классификаторов: точность, полнота, безошибочность и кривые ROC

В задаче бинарной классификации выдается одно из двух значений: 0/1, да/нет и т. д. В более абстрактном смысле речь идет о позитивных и негативных результатах. Возвращая предсказание, наша сеть либо права, либо ошибается, так что существует четыре возможных сочетания настоящей метки входного примера данных и выданного сетью значения, как демонстрирует табл. 3.1.

Таблица 3.1. Четыре вида результатов классификации при задаче бинарной классификации

		Предсказание	
		Позитивный	Негативный
Фактический	Позитивный	Истиннопозитивный (TP)	Ложнонегативный (FN)
	Негативный	Ложнопозитивный (FP)	Истиннонегативный (TN)

Истиннопозитивные и истиннонегативные результаты — те, где модель выдала правильное предсказание; ложнопозитивные и ложнонегативные результаты — те, где модель ошиблась. Если занести во все четыре ячейки количества, получится *матрица различий* (confusion matrix). В табл. 3.2 приведена гипотетическая матрица различий для нашей задачи обнаружения фишинговых сайтов.

Таблица 3.2. Матрица различий для гипотетической задачи бинарной классификации

		Предсказание	
		Позитивный	Негативный
Фактический	Позитивный	4	2
	Негативный	1	93

В наших гипотетических результатах на примерах для фишинговых сайтов правильно идентифицированы четыре фишинговые веб-страницы, пропущено две и допущено одно ложное оповещение. Рассмотрим теперь другие распространенные метрики, отражающие те же показатели работы модели.

Простейшая из метрик — *безошибочность* (accuracy). Она количественно отражает долю верно классифицированных примеров данных:

```
безошибочность = (#TP + #TN) / #примеров_данных = (#TP + #TN) /
(#TP + #TN + #FP + #FN)
```

В нашем конкретном примере:

```
безошибочность = (4 + 93) / 100 = 97%
```

Безошибочность — простой и понятный показатель. Впрочем, иногда он может вводить в заблуждение — в задачах бинарной классификации позитивные и не-

гативные примеры данных зачастую не распределены равным образом. Нередко встречается ситуация, когда количество позитивных примеров данных намного меньше, чем негативных (например, большинство ссылок — не фишинговые, большинство деталей — не бракованные и т. д.). А если только 5 ссылок из 100 фишинговые, наша сеть может всегда выдавать, что ссылка не фишинговая, и безошибочность будет равна 95 %! С этой точки зрения безошибочность — очень плохая мера качества работы системы. Всегда кажется, что высокая безошибочность — это хорошо, но часто такой вывод становится преждевременным. Следить за показателем безошибочности не помешает, но в качестве функции потерь она подходит плохо.

Следующая пара метрик, улавливающих нюансы ошибок, допускаемых моделью: *точность* (precision) и *полнота* (recall). В дальнейшем мы будем обсуждать задачи, в которых позитивный результат подразумевает последующие действия — подсветку ссылки, пометку сообщения для дальнейшего просмотра в ручном режиме, в то время как негативный результат означает, что ничего не надо делать. Эти метрики относятся к различным видам «неправильности» предсказаний нашей модели.

Точность — отношение количества позитивных предсказаний модели к действительно позитивным:

```
точность = #TP / (#TP + #FP)
```

При значениях, указанных в нашей матрице различий, получается:

```
точность = 4 / (4 + 1) = 80%
```

Как и безошибочность, показатель точности можно обмануть. Можно маркировать как позитивные только входные примеры данных лишь с очень большим выходным сигналом сигма-функции (скажем, > 0,95 вместо > 0,5 по умолчанию), сделав модель очень консервативной. Обычно подобный вариант приводит к росту точности, но и к пропуску моделью множества действительно позитивных примеров данных (маркированию их как негативных). Эту проблему позволяет решить еще одна метрика, часто сопутствующая точности и дополняющая ее, — полнота.

Полнота — отношение количества действительно позитивных примеров данных к количеству примеров, классифицированных моделью как позитивные:

```
полнота = #TP / (#TP + #FN)
```

Для нашего примера получаем:

```
полнота = 4 / (4 + 2) = 66.7%
```

Сколько же данных набора нашла модель из всех позитивных примеров? Обычно принимается осознанное решение повысить частоту ложных срабатываний, чтобы снизить вероятность пропустить что-либо. Чтобы обмануть эту метрику, достаточно объявить все примеры данных позитивными; поскольку ложнопозитивные примеры не включаются в уравнение, получится 100%-ная полнота за счет снижения точности.

Как видим, довольно просто создать систему, демонстрирующую отличные результаты относительно метрики безошибочности, полноты или точности. На практике же в задачах бинарной классификации зачастую очень непросто достичь

одновременно высокой точности и полноты (в ином случае задача оказалась бы очень простой и машинное обучение вообще бы не понадобилось). Точность и полнота служат для тонкой подстройки модели в областях, где принципиально неизвестно, каким должен быть правильный ответ. Вы встретите еще более дифференцированные и сложные метрики, например *Точность при полноте X %*, где X около 90 % — точность при настройке модели на поиск как минимум 90 % позитивных примеров данных. Например, из рис. 3.5 видно, что после 400 эпох обучения наша модель обнаружения фишинга достигла точности 96,8 %, а полноты — в 92,9 % при пороговом значении вероятности классификации 0,5.

Важно отметить, что пороговое значение выходного сигнала сигма-функции для выбора позитивных предсказаний не должно быть равно 0,5. На самом деле в зависимости от обстоятельств может быть удобнее задать его равным значению больше 0,5 (но меньше 1) или меньше 0,5 (но больше 0). При снижении этого порогового значения модель с большей щедростью раздает входным примерам данных позитивные метки, в результате чего повышается полнота, но снижается точность. С другой стороны, при повышении этого порогового значения модель с большей осторожностью раздает входным примерам данных позитивные метки, так что точность повышается, зато снижается полнота. Следовательно, наблюдается определенный баланс между точностью и полнотой, который весьма непросто выразить количественно с помощью какой-либо из вышеупомянутых метрик. К счастью, обширные исследования в области бинарной классификации выявили лучшие способы количественного выражения и визуализации этого соотношения. Для данной цели часто используется кривая ROC.

3.2.3. Кривая ROC: наглядное представление соотношения плюсов и минусов при бинарной классификации

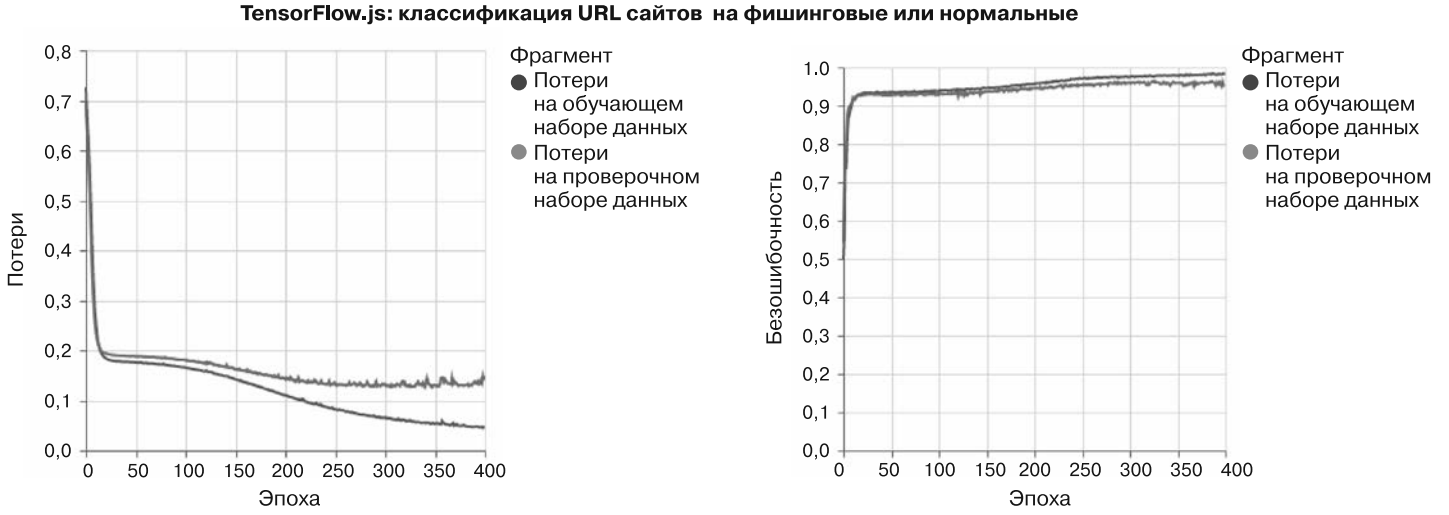
Кривые ROC применяются во множестве разнообразных инженерных задач, требующих бинарной классификации или обнаружения определенных видов событий. Полное ее название — *рабочие характеристики приемника* (receiver operating characteristic) — появилось в эпоху первых радаров. Сегодня оно практически не встречается в литературе. На рис. 3.6 приведен пример кривой ROC для нашего приложения.

Как видно из меток осей координат на рис. 3.6, кривые ROC не отражают зависимость точности и полноты, а основаны на двух несколько других метриках. По горизонтальной оси координат кривой ROC откладывается *частота ложнопозитивных* результатов классификации (false positive rate, FPR):

```
FPR = #FP / (#FP + #TN)
```

А по вертикальной оси кривой ROC откладывается *частота истиннопозитивных* результатов (true positive rate, TPR):

```
TPR = #TP / (#TP + #FN) = полнота
```



Итоговые потери на обучающем наборе данных: 0,0493; безошибочность: 0,9801
Итоговые потери на проверочном наборе данных: 0,1402; безошибочность: 0,9521
Потери на контрольном наборе данных: 0,1317; безошибочность: 0,9555
Точность: 0,9675
Полнота: 0,9289
Частота ложнопозитивных результатов (FPR): 0,0240
Площадь под кривой (AUC): 0,981150715602107

Рис. 3.5. Пример прохождения цикла обучения модели для обнаружения фишинговых сайтов. Обратите внимание на различные метрики внизу рисунка: точность, полнота и FPR. Метрика «Площадь под кривой» (area under the curve, AUC) обсуждается в подразделе 3.2.3

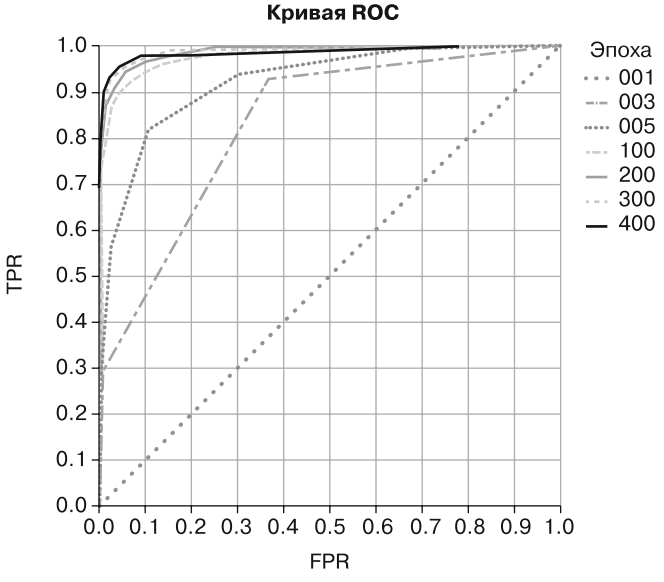


Рис. 3.6. Набор примеров кривых ROC, полученных во время обучения модели для обнаружения фишинга. Все кривые относятся к различным эпохам и демонстрируют постепенное улучшение качества модели бинарной классификации по мере ее обучения

Определение TPR соответствует определению полноты — это просто другое название метрики. FPR же представляет нечто новое для нас. В знаменателе ее формулы — количество всех случаев, когда на самом деле класс примера данных негативный; числитель же равен количеству всех ложнопозитивных случаев. Другими словами, FPR — это доля действительно негативных примеров данных, ошибочно классифицированных как позитивные, то есть вероятность так называемого *ложного срабатывания* (false alarm). В табл. 3.3 приведен список наиболее часто встречающихся метрик для задач бинарной классификации.

Таблица 3.3. Распространенные метрики для задач бинарной классификации

Название метрики	**Определение**	**Использование в кривых ROC или кривых зависимости «точность/полнота»**
Безошибочность	(#TP + #TN) / (#TP + + #TN + # FP + #FN)	Не используется в кривых ROC
Точность	#TP / (#TP + #FP)	Вертикальная ось координат графика кривой «точность/полнота»
Полнота/чувствительность/частота истиннопозитивных результатов (TPR)	#TP / (#TP + #FN)	Вертикальная ось координат графика кривой ROC (как на рис. 3.6) или горизонтальная ось координат графика кривой «точность/полнота»
Частота ложнопозитивных результатов (FPR)	#FP / (#FP + #TN)	Горизонтальная ось координат графика кривой ROC (см. рис. 3.6)

Название метрики	Определение	Использование в кривых ROC или кривых зависимости «точность/полнота»
Площадь под кривой (AUC)	Вычисляется путем численного интегрирования площади под кривой ROC; см. пример в листинге 3.7	Не используется в кривых ROC, но вычисляется на основе кривых ROC

Семь кривых ROC на рис. 3.6 нарисованы в начале семи различных эпох: от первой (001) до последней (400). Все они созданы на основе предсказаний модели на контрольном (а не обучающем) наборе данных. В листинге 3.6 приводятся подробности того, как это было сделано, с помощью обратного вызова `onEpochBegin` API `Model.fit()`. Благодаря этому подходу можно выполнить интересный анализ и визуализацию модели во время обучения без необходимости писать цикл `for` или использовать несколько вызовов `Model.fit()`.

Листинг 3.6. Визуализация кривых ROC во время обучения модели с помощью обратного вызова

```
await model.fit(trainData.data, trainData.target, {
  batchSize,
  epochs,
  validationSplit: 0.2,
  callbacks: {
  onEpochBegin: async (epoch) => {
      if ((epoch + 1)% 100 === 0 ||
          epoch === 0 || epoch === 2 || epoch === 4) {
          const probs = model.predict(testData.data);
          drawROC(testData.target, probs, epoch);
      }
    },
    onEpochEnd: async (epoch, logs) => {
      await ui.updateStatus(
         `Epoch ${epoch + 1} of ${epochs} completed.`);
      trainLogs.push(logs);
      ui.plotLosses(trainLogs);
      ui.plotAccuracies(trainLogs);
    }
  }
});
```

Рисуем кривую ROC через каждые несколько эпох

Подробности того, как рисуется кривая ROC, содержатся в функции `drawROC()` (листинг 3.7). Она делает следующее.

- Варьирует пороговое значение выходного сигнала сигма-функции (вероятностей) нейронной сети для получения различных наборов результатов классификации.
- Вычисляет TPR и FPR для каждого результата классификации, сопоставляя его с фактическими (целевыми) метками.
- Строит график зависимости TPR и FPR, формируя кривую ROC.

Как видно на рис. 3.6, в начале обучения (эпоха 001) в силу инициализации весовых коэффициентов модели случайными числами кривая ROC очень близка к диагонали, соединяющей точку (0,0) с точкой (1,1). По мере обучения кривые ROC все больше прижимаются к верхнему левому углу — месту, где показатель FPR близок к 0, а TPR — к 1. При любом конкретном уровне FPR, например 0,1, наблюдается монотонное возрастание соответствующего значения TPR по мере обучения. Проще говоря, по мере обучения можно получать все более и более высокий уровень полноты (TPR) при фиксированном уровне ложных срабатываний (FPR).

«Идеальной» ROC является кривая, изогнутая в сторону верхнего левого угла настолько, что она приобретает γ-образную форму. При таком сценарии достигается 100%-ная TPR и 0%-ная FPR — заветная мечта любого бинарного классификатора. На практике, однако, можно лишь подталкивать кривую ROC как можно ближе к верхнему левому углу — теоретический идеал верхнего левого угла недостижим.

Из этого обсуждения формы кривой ROC видно: можно определить, насколько хороша кривая ROC, просто по площади под ней, то есть площади в квадратных единицах, ограничиваемой кривой ROC и осью координат *X*. Этот показатель называется *площадью под кривой* (area under the curve, AUC) и также рассчитывается в коде из листинга 3.7. Это более удобная метрика, чем точность, полнота и безошибочность, — она учитывает баланс между ложнопозитивными и ложнонегативными результатами. AUC кривой ROC для случайного угадывания (диагональ) равна 0,5, в то время как у идеальной ROC γ-образной формы AUC равна 1,0. После обучения наша модель обнаружения фишинга достигает AUC 0,981.

Помимо визуализации характеристик бинарного классификатора, кривая ROC помогает обоснованно выбрать пороговое значение вероятности в реальных условиях. Например, представьте себе коммерческую компанию, разрабатывающую детектор фишинга в качестве сервиса. Какой же из вариантов лучше выбрать?

- Относительно низкое пороговое значение, поскольку пропуск действительно фишингового сайта может привести к серьезным неприятностям и потерянным контрактам.
- Относительно высокое пороговое значение, поскольку нас больше беспокоят возможные жалобы пользователей, чьи нормальные сайты оказались заблокированы из-за того, что модель ошибочно классифицировала их как фишинговые.

Каждому пороговому значению соответствует точка на кривой ROC. При постепенном повышении порогового значения от 0 до 1 мы перемещаемся из верхнего правого угла графика (где FPR и TPR равны 1) в нижний левый (где FPR и TPR равны 0). В реальных инженерных задачах точка на кривой ROC всегда выбирается исходя из баланса подобных противоположных затрат и может варьироваться для различных клиентов и фаз развития бизнеса.

Помимо кривой ROC, для визуализации бинарной классификации часто применяется *кривая «точность/полнота»* (которую иногда называют кривой P/R), мельком упомянутая в табл. 3.3. В отличие от кривой ROC, график «точность/полнота» отражает зависимость точности от полноты. Поскольку кривые «точность/полнота» по своей сути не отличаются от кривых ROC, мы не станем обсуждать их здесь.

Листинг 3.7. Код вычисления и визуализации кривой ROC и ее AUC

```
function drawROC(targets, probs, epoch) {
  return tf.tidy(() => {
    const thresholds = [
      0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45,
      0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85,
      0.9, 0.92, 0.94, 0.96, 0.98, 1.0
    ];
    const tprs = [];  // Ложнопозитивные результаты
    const fprs = [];  // Ложнонегативные результаты
    let area = 0;
    for (let i = 0; i < thresholds.length; ++i) {
           const threshold = thresholds[i];
      const threshPredictions =
            utils.binarize(probs, threshold).as1D();
      const fpr = falsePositiveRate(
                  targets,
      threshPredictions).arraySync();
      const tpr = tf.metrics.recall(targets, threshPredictions).arraySync();
      fprs.push(fpr);
      tprs.push(tpr);

      if (i > 0) {
        area += (tprs[i] + tprs[i - 1]) * (fprs[i - 1] - fprs[i]) / 2;
      }
    }
    ui.plotROC(fprs, tprs, epoch);
    return area;
  });
}
```

Выбираемый вручную набор пороговых значений вероятности

Функция falsePositiveRate() вычисляет частоту ложнопозитивных результатов путем сравнения предсказаний и фактических целевых меток. Она описана в том же файле

Преобразуем вероятность в предсказания, пропуская через сито порогового значения

Интегрирование для вычисления AUC

Стоит отметить в листинге 3.7 вызов функции `tf.tidy()`. Она гарантирует, что память после тензоров, созданных внутри переданной в нее анонимной функции, будет должным образом освобождена и они не будут занимать память WebGL. В браузере TensorFlow.js не может повлиять на выделяемую для пользовательских тензоров память в основном из-за отсутствия финализации объектов в JavaScript и сборки мусора для текстур WebGL, лежащих в основе тензоров TensorFlow.js. Если не очистить должным образом подобные промежуточные тензоры, произойдет утечка памяти WebGL. А если такие утечки памяти будут продолжаться в течение долгого времени, то в конце концов они приведут к ошибкам нехватки памяти WebGL. Раздел Б.3 содержит подробное руководство по управлению памятью в TensorFlow.js. А в разделе Б.5 вы найдете упражнения на эту тему. Внимательно изучите эти разделы, если собираетесь описывать пользовательские функции на основе композиции функций TensorFlow.js.

3.2.4. Бинарная перекрестная энтропия

До сих пор мы говорили о нескольких различных метриках, количественно выражающих разные аспекты качества работы бинарного классификатора, в частности о безошибочности, точности и полноте (см. табл. 3.3). Но мы не затрагивали важную

метрику, дифференцируемую и способную генерировать градиенты, подходящие для обучения модели на основе градиентного спуска. Мы говорим о `binaryCrossentropy`, мельком упомянутой в листинге 3.5:

```
model.compile({
  optimizer: 'adam',
  loss: 'binaryCrossentropy',
  metrics: ['accuracy']
});
```

Прежде всего вы можете задать вопрос: почему просто не взять безошибочность, точность, полноту или даже, например, AUC и не использовать их в качестве функции потерь? В конце концов, эти метрики вполне понятны. Кроме того, в задачах регрессии выше мы использовали в качестве функции потерь MSE, тоже понятную метрику. Дело в том, что ни одна из этих метрик бинарной классификации не дает необходимых для обучения градиентов. Возьмем, например, безошибочность: чтобы понять, почему она не подходит для генерирования градиентов, достаточно осознать, что для ее вычисления необходимо определить, какие из предсказаний модели позитивные, а какие — негативные (см. первую строку табл. 3.3). Для этого нужно воспользоваться *пороговой функцией* (thresholding function), которая бы преобразовывала выходной сигнал сигма-функции модели в бинарные предсказания. В этом и кроется корень проблемы: хотя пороговая функция (*ступенчатая функция*, говоря более строгим языком) дифференцируема почти везде («почти», поскольку она не дифференцируема в «точке скачка» 0,5), ее производная везде равна нулю (рис. 3.7)! Что же будет, если попытаться произвести обратное распространение ошибки через такую пороговую функцию? Все градиенты в конце концов превратятся в нули, поскольку в определенный момент значения градиентов умножаются на нулевые производные ступенчатой функции. Проще говоря, если в качестве функции потерь используется безошибочность (точность, полнота, AUC

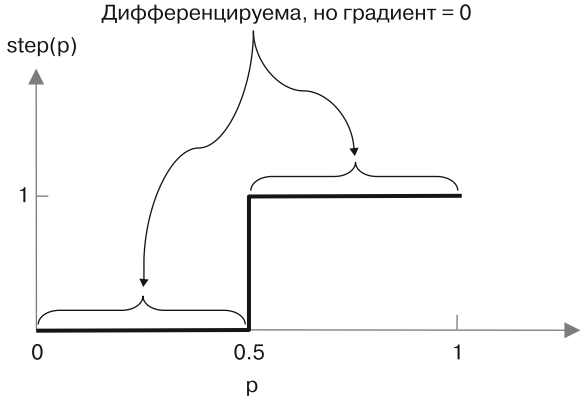


Рис. 3.7. Ступенчатая функция, используемая для преобразования вероятности, получаемой на выходе модели бинарной классификации, дифференцируема почти везде. К сожалению, градиент (производная) во всех точках, где она дифференцируема, равен нулю

и т. д.), из-за плоских участков графика ступенчатой функции, лежащей в ее основе, процедура обучения не может определить, куда нужно двигаться в пространстве весов, чтобы снизить потери.

Следовательно, при безошибочности в качестве функции потерь мы не сможем вычислить пригодные для нас градиенты, а значит, и найти осмысленные обновления для весовых коэффициентов модели. Те же ограничения относятся и к таким метрикам, как точность, полнота, FPR и AUC. И хотя с помощью этих метрик людям удобно разбираться в поведении бинарного классификатора, для процесса обучения моделей они бесполезны.

Для задачи бинарной классификации воспользуемся в качестве функции потерь *бинарной перекрестной энтропией* (binary cross entropy), соответствующей значению параметра `'binaryCrossentropy'` в коде нашей модели для обнаружения фишинга (см. листинги 3.5 и 3.6). Алгоритмически можно описать бинарную перекрестную энтропию с помощью такого псевдокода (листинг 3.8).

Листинг 3.8. Псевдокод для функции потерь на основе бинарной перекрестной энтропии[1]

```
function binaryCrossentropy(truthLabel, prob):
  if truthLabel is 1:
   return -log(prob)
  else:
   return -log(1 - prob)
```

В этом псевдокоде `truthLabel` может принимать значения 0 или 1 и указывает, какая на самом деле метка у входного примера — позитивная или негативная. `prob` — вероятность принадлежности примера к позитивному классу, предсказанная моделью. Обратите внимание, что, в отличие от `truthLabel`, `prob` является вещественным числом, которое может принимать любое значение от 0 до 1. `log` — натуральный алгоритм по основанию e (= 2,718), как вы помните из школьного курса математики. В теле функции `binaryCrossentropy` содержится условный оператор `if-else`, и она выполняет различные действия в зависимости от того, чему равно `truthLabel` — 0 или 1. На рис. 3.8 на одном графике показаны оба случая.

Глядя на рис. 3.8, учтите, что чем меньше значение, тем лучше, ведь это функция потерь. Относительно этой функции потерь важно отметить следующее.

- Если `truthLabel` равно 1, то чем ближе `prob` к 1,0, тем ниже значение функции потерь. Это вполне логично, ведь для фактически позитивного примера данных модель должна выдавать как можно более близкую к 1,0 вероятность. И напротив, если `truthLabel` равно 0, то чем ближе вероятность к 0, тем ниже значение функции потерь. Это логично, ведь в таком случае модель должна выдавать как можно более близкую к 0 вероятность.

[1] В реальном коде для binaryCrossentropy необходимо предусмотреть случаи, когда prob или 1 – prob в точности равны нулю, при передаче которого в функцию log возникла бы бесконечность. Для решения этой проблемы к prob и 1 – prob перед передачей их в функцию логарифма прибавляется очень маленькое число (например, 1e-6), обычно называемое «эпсилон» или «поправочный параметр».

- В отличие от бинарной пороговой функции, приведенной на рис. 3.7, угловые коэффициенты этих кривых в каждой точке ненулевые, вследствие чего градиенты также ненулевые. Это подходит для обучения модели на основе метода обратного распространения ошибки.

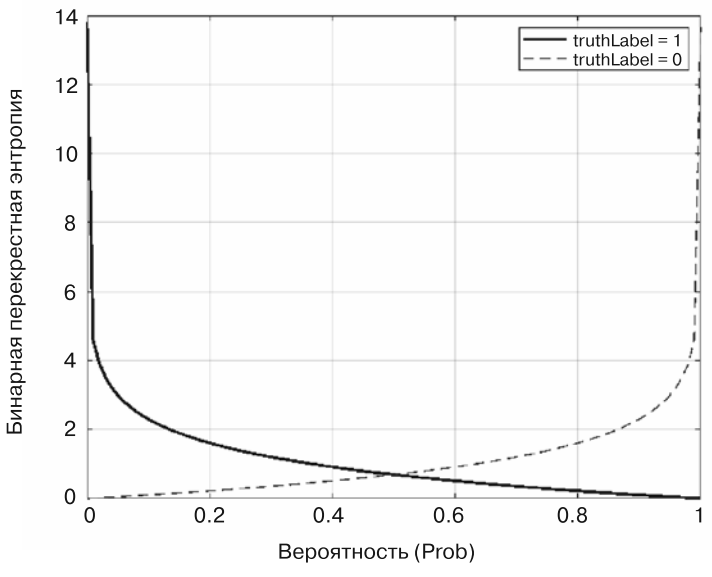


Рис. 3.8. Функция потерь бинарной перекрестной энтропии. Отдельно приведены графики двух случаев (truthLabel = 1 и truthLabel = 0), отражая оператор логического ветвления if-else в листинге 3.8

Вы можете задаться вопросом, почему не повторить то, что мы проделали для регрессионной модели: просто притвориться, что значения 0–1 — целевые переменные регрессии, и воспользоваться MSE в качестве функции потерь? В конце концов, MSE дифференцируема, а вычисление MSE между меткой истины и вероятностью даст ненулевые производные аналогично `binaryCrossentropy`. Проблема в снижении «отдачи» MSE на границах. Например, в табл. 3.4 перечислены значения функции потерь `binaryCrossentropy` и MSE для нескольких значений `prop` при `truthLabel = 1`. По мере приближения `prob` к 1 (желаемому значению) MSE снижается все медленнее, по сравнению с `binaryCrossentropy`. В результате она плохо «стимулирует» выдачу моделью более высоких (близких к 1) значений `prob`, когда `prob` уже и так довольно близка к 1 (например, равна 0,9). Аналогично, когда `truthLabel = 0`, MSE хуже подходит, чем `binaryCrossentropy`, для генерации градиентов, подталкивающих выходной сигнал `prob` модели в сторону нуля.

Это демонстрирует, что задачи бинарной классификации отличаются от задач регрессии по крайней мере еще в одном аспекте: в первых потери (`binaryCrossentropy`) и метрики (безошибочность, точность и т. д.) различны, а во вторых обычно одинаковы (например, `meanSquaredError`). Как мы увидим в следующем разделе, в задачах многоклассовой классификации функции потерь и метрики также отличаются.

Таблица 3.4. Сравнение значений бинарной перекрестной энтропии и MSE для гипотетических результатов бинарной классификации

truthLabel	prob	Бинарная перекрестная энтропия	MSE
1	0,1	2,302	0,81
1	0,5	0,693	0,25
1	0,9	0,100	0,01
1	0,99	0,010	0,0001
1	0,999	0,001	0,000001
1	1	0	0

3.3. Многоклассовая классификация

В разделе 3.2 мы научились структурировать задачи бинарной классификации. Теперь же вас ждет небольшое отступление: поговорим о *небинарной классификации* (nonbinary classification) — задачах классификации с тремя или более классами[1]. Для иллюстрации многоклассовой классификации воспользуемся известным набором данных «Ирисы Фишера», пришедшим к нам из области статистики (https://ru.wikipedia.org/wiki/Ирисы_Фишера). Он состоит из данных об ирисах трех видов: ирис щетинистый *(Iris setosa)*, ирис виргинский *(Iris virginica)* и ирис разноцветный *(Iris versicolor)*. Эти три вида можно отличить друг от друга по форме и размерам. В начале XX столетия британский статистик Рональд Фишер измерил длину и ширину лепестков и чашелистиков (различных частей цветка) 150 экземпляров ирисов. Набор данных сбалансирован: по 50 примеров для каждой целевой метки.

В этой задаче модель принимает четыре числовых входных признака — длину лепестка, ширину лепестка, длину чашелистика, ширину чашелистика — и пытается предсказать целевую метку (один из трех видов цветов). Этот пример вы можете найти в каталоге `iris` репозитория tfjs-examples. Извлечь его и запустить можно с помощью следующих команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/iris
yarn && yarn watch
```

[1] Не путайте многоклассовую (multiclass) классификацию с многозначной (multilabel, буквально «с несколькими метками»). При многозначной классификации отдельному входному примеру данных может соответствовать несколько выходных классов. В качестве примера можно привести задачу обнаружения различных типов объектов во входном изображении. На одном может быть изображен только один человек; а другое может включать изображения людей, машин и животных. Многозначный классификатор должен выдавать на выходе результаты, отражающие все классы, применимые к входному примеру, в зависимости от того, один такой класс или несколько. В этом разделе мы не станем обсуждать многозначную классификацию, а поговорим о простой многоклассовой классификации с одной меткой, в которой каждому входному примеру соответствует ровно один из двух или более возможных выходных классов.

3.3.1. Унитарное кодирование категориальных данных

Прежде чем изучать модель, предназначенную для решения задачи классификации ирисов, необходимо поговорить о способе представления целевых меток (видов цветов) в этой задаче многоклассовой классификации. Во всех предыдущих примерах машинного обучения в книге представление целевых признаков было более простым, например в виде одного числа в задаче предсказания времени скачивания и задаче предсказания цен на бостонскую недвижимость, а также представления 0-1 бинарных целевых признаков в задаче обнаружения фишинга. В задаче же классификации ирисов три вида цветков представлены несколько менее привычным способом, с помощью так называемого *унитарного кодирования* (one-hot encoding). Откройте файл `data.js` и взгляните на строку:

```
const ys = tf.oneHot(tf.tensor1d(shuffledTargets).toInt(), IRIS_NUM_CLASSES);
```

Здесь `shuffledTargets` — простой JavaScript-массив, состоящий из целочисленных меток для примеров в перетасованном виде. Значения всех его элементов равны 0, 1 или 2: соответственно трем видам ирисов в наборе данных. С помощью вызова `tf.tensor1d(shuffledTargets).toInt()` набор преобразуется в одномерный тензор с типом элементов `int32`. Затем он передается в функцию `tf.oneHot()`, возвращающую двумерный тензор формы `[numExamples, IRIS_NUM_CLASSES]`. `numExamples` — число примеров данных, содержащихся в `targets`, а `IRIS_NUM_CLASSES` — просто константа, равная 3. Чтобы взглянуть на фактические значения `targets` и `ys`, можете добавить сразу после вышеупомянутой строки код для вывода в консоль наподобие следующего:

```
const ys = tf.oneHot(tf.tensor1d(shuffledTargets).toInt(), IRIS_NUM_CLASSES);
// Добавленные строки кода для вывода в консоль значений `targets` и `ys`
console.log('Value of targets:', targets);
ys.print();[1]
```

После этих изменений процесс-упаковщик Parcel, запускаемый Yarn-командой `watch` в терминале, автоматически пересобирает веб-файлы. Далее вы можете открыть инструменты разработчика (devtool) на соответствующей вкладке браузера и обновить страницу. Сообщения, выводимые вызовами `console.log()` и `print()`, попадают в консоль devtool. Они выглядят примерно так:

```
Value of targets: (50) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0]

Tensor
    [[1, 0, 0],
```

[1] В отличие от targets, ys не просто JavaScript-массив, а тензорный объект, использующий память GPU. Следовательно, с помощью обычного вызова console.log его значение не посмотреть. Метод print() специально предназначен для извлечения значений из GPU, их форматирования с учетом формы тензора в удобном для человека виде и вывода в консоль.

```
     [1, 0, 0],
     [1, 0, 0],
     ...,
     [1, 0, 0],
     [1, 0, 0],
     [1, 0, 0]]
```

или

```
Value of targets: (50) [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1]

Tensor
    [[0, 1, 0],
     [0, 1, 0],
     [0, 1, 0],
     ...,
     [0, 1, 0],
     [0, 1, 0],
     [0, 1, 0]]
```

и т. д. Иными словами, примеру данных с числовой меткой 0 соответствует строка значений `[1, 0, 0]`; примеру данных с числовой меткой 1 соответствует строка значений `[0, 1, 0]` и т. д. Это простой и понятный пример унитарного кодирования: целочисленная метка превращается в вектор, состоящий из всех нулей и одной единицы — на позиции, соответствующей этой метке. Длина вектора равна количеству всех возможных категорий. Именно потому, что в векторе содержится только одно значение `1`, эта схема кодирования и называется унитарной.

Возможно, такое кодирование покажется вам слишком запутанным. Зачем использовать три числа в качестве представления категории, когда достаточно одного? Почему этот вариант кодирования предпочтительнее более простого и экономичного кодирования с одним целочисленным индексом? Чтобы разобраться, стоит взглянуть на эту схему кодирования с двух различных сторон.

Во-первых, нейронной сети гораздо проще выдавать непрерывные вещественные значения, а не целочисленные. Округление выходного сигнала вещественного типа также представляется не слишком изящным решением. Намного более изящно и естественно будет, если последний слой нейронной сети станет выдавать несколько отдельных вещественных чисел, ограниченных интервалом `[0, 1]`, благодаря специально подобранной функции наподобие сигма-функции активации, которую мы использовали для бинарной классификации. При таком подходе каждое из чисел отражает оценку моделью вероятности того, что входной пример принадлежит к соответствующему классу. Именно для этого и предназначено унитарное кодирование: оно представляет собой «правильный ответ» для оценок вероятностей, на который модель должна ориентироваться в процессе обучения.

Во-вторых, благодаря целочисленному кодированию категорий классы неявным образом упорядочиваются. Например, можно задать для ириса щетинистого метку 0, для ириса разноцветного — метку 1, а для ириса виргинского — метку 2. Но подобные схемы упорядочения зачастую выглядят неестественно и их выбор

кажется неоправданным. Например, такая схема нумерации подразумевает, что ирис щетинистый ближе к ирису разноцветному, чем к виргинскому, что вовсе не обязательно соответствует действительности. Нейронные сети работают с вещественными числами и основаны на таких математических операциях, как умножение и сложение. Следовательно, они чувствительны к порядку чисел и их упорядоченности. Если кодировать категорию одним числом, возникает дополнительная нелинейная связь, которую нейронной сети необходимо усвоить. И напротив, унитарно закодированные категории не подразумевают никакого неявного упорядочения, а потому не предъявляют подобных лишних требований к усвоению нейронной сетью информации.

В главе 9 вы узнаете, что унитарное кодирование применяется не только для выходных целевых признаков нейронных сетей, но и в тех случаях, когда на вход нейронных сетей поступают категориальные данные.

3.3.2. Многомерная логистическая функция активации

Теперь, когда мы разобрались с представлением входных признаков и выходных целей, можно взглянуть на код описания модели (из файла `iris/index.js`).

Листинг 3.9. Многослойная нейронная сеть для классификации ирисов

```
const model = tf.sequential();
model.add(tf.layers.dense(
    {units: 10, activation: 'sigmoid', inputShape: [xTrain.shape[1]]}));
model.add(tf.layers.dense({units: 3, activation: 'softmax'}));
model.summary();

const optimizer = tf.train.adam(params.learningRate);
model.compile({
  optimizer: optimizer,
  loss: 'categoricalCrossentropy',
  metrics: ['accuracy'],
});
```

Описанная в листинге 3.9 модель выдает следующие результаты:

```
_________________________________________________________________
Layer (type)                 Output shape              Param #
=================================================================
dense_Dense1 (Dense)         [null,10]                 50
_________________________________________________________________
dense_Dense2 (Dense)         [null,3]                  33
=================================================================
Total params: 83
Trainable params: 83
Non-trainable params:
_________________________________________________________________
```

Как видно из вывода, это достаточно простая модель с относительно небольшим количеством (83) весовых коэффициентов. Форма выходного сигнала `[null, 3]` соответствует унитарному представлению категориального целевого признака. Используемая для последнего слоя функция активации, а именно *многомерная логистическая функция* (softmax), создана специально для задач многоклассовой классификации. Математическое определение многомерной логистической функции можно записать в псевдокоде следующим образом:

```
softmax([x1, x2, …, xn]) =
    [exp(x1) / (exp(x1) + exp(x2) + … + exp(xn)),
     exp(x2) / (exp(x1) + exp(x2) + … + exp(xn)),
     ...,
     exp(xn) / (exp(x1) + exp(x2) + … + exp(xn))]
```

В отличие от сигма-функции активации многомерная логистическая функция активации применяется не поэлементно, преобразование каждого из элементов входного вектора зависит от всех остальных элементов. А именно: берется экспонента каждого из элементов входного вектора (функция `exp` с основанием $e = 2{,}718$). Полученное делится на сумму экспонент всех элементов. Что это нам дает? Во-первых, гарантирует, что все числа находятся в интервале от 0 до 1. Во-вторых, означает, что сумма всех элементов выходного вектора равна 1. Такое свойство желательно, поскольку: 1) выходные сигналы можно интерпретировать как соответствующие классам оценки вероятностей и 2) выходные сигналы должны удовлетворять этому свойству для совместимости с категориальной перекрестной энтропией в качестве функции потерь. В-третьих, это определение гарантирует, что больший элемент во входном векторе соответствует большему элементу в выходном. Для примера, допустим, что в результате умножения на матрицу и прибавления смещения в последнем плотном слое получается такой вектор:

```
[-3, 0, -8]
```

Его длина равна 3, поскольку плотный слой содержит три нейрона. Обратите внимание, что элементы представляют собой вещественные числа, не ограниченные никаким диапазоном. Многомерная логистическая функция активации преобразует этот вектор в такой:

```
[0.0474107, 0.9522698, 0.0003195]
```

Можете проверить это сами, запустив следующий код TensorFlow.js (например, в консоли инструментов разработчика на странице js.tensorflow.org):

```
const x = tf.tensor1d([-3, 0, -8]);
tf.softmax(x).print();
```

Три элемента из результата многомерной логистической функции: 1) находятся в интервале `[0, 1]`, 2) равны в сумме 1 и 3) упорядочены таким же образом, как и элементы входного вектора. Благодаря этим свойствам выходной сигнал можно интерпретировать как вероятности, которые модель назначает всем возможным классам. В предыдущем фрагменте кода наибольшая вероятность соответствует второй категории, а наименьшая — первой.

Следовательно, при использовании подобного многоклассового классификатора принадлежность элемента к тому или иному классу можно выбирать по номеру выходного вектора функции softmax с максимальным значением. Для этого можно воспользоваться методом `argMax()`. Например, вот фрагмент файла `index.js`:

```
const predictOut = model.predict(input);
const winner = data.IRIS_CLASSES[predictOut.argMax(-1).dataSync()[0]];
```

`predictOut` — двумерный тензор формы `[numExamples, 3]`. При вызове его метода `argMax()` форма свертывается в `[numExamples]`. Значение аргумента `-1` указывает методу `argMax()` искать максимальные значения по последнему измерению и возвращать их индексы. Например, пусть значение `predictOut` равно:

```
[[0  , 0.6, 0.4],
 [0.8, 0  , 0.2]]
```

Далее, `argMax(-1)` возвращает тензор, согласно которому максимальные значения по последнему измерению находятся на позициях 1 и 0 для первого и второго примера данных соответственно:

```
[1, 0]
```

3.3.3. Категориальная перекрестная энтропия: функция потерь для многоклассовой классификации

В примере бинарной классификации мы видели, как в качестве функции потерь использовалась бинарная перекрестная энтропия и почему другие, более понятные человеку метрики, например безошибочность и полнота, не подходят на роль функции потерь.

При многоклассовой классификации ситуация аналогична. Для нее существует простая и понятная метрика — безошибочность — доля правильно классифицированных моделью примеров данных. Эта метрика дает возможность специалистам, использующим модель, понять, насколько хорошо та работает. Она используется в следующем фрагменте кода из листинга 3.9:

```
model.compile({
  optimizer: optimizer,
  loss: 'categoricalCrossentropy',
  metrics: ['accuracy'],
 });
```

Однако безошибочность — плохой кандидат на роль функции потерь, поскольку эта метрика подвержена той же проблеме нулевых градиентов, что и безошибочность при бинарной классификации. Следовательно, пришлось изобрести специальную функцию потерь для многоклассовой классификации: *категориальную перекрестную энтропию* (categorical cross entropy). Это просто обобщение бинарной перекрестной энтропии на случай более чем двух категорий.

Листинг 3.10. Псевдокод функции потерь на основе категориальной перекрестной энтропии

```
function categoricalCrossentropy(oneHotTruth, probs):
  for i in (0 to length of oneHotTruth)
    if oneHotTruth(i) is equal to 1
      return -log(probs[i]);
```

В псевдокоде из предыдущего листинга `oneHotTruth` — унитарное представление фактического класса входного примера, а `probs` — вероятности на выходе многомерной логистической функции модели. Основной вывод из этого псевдокода — с точки зрения категориальной перекрестной энтропии важен лишь один элемент `probs`, а именно тот, индексы которого соответствуют фактическому классу. Остальные элементы `probs` могут быть какими угодно, но если они не изменяют значение элемента для фактического класса, то на категориальную перекрестную энтропию не влияют. Что же касается этого конкретного элемента `probs`, то чем ближе он к 1, тем меньше будет значение перекрестной энтропии. Как и для бинарной перекрестной энтропии, для категориальной существует функция в пространстве имен `tf.metrics`, с помощью которой вы можете вычислить категориальную перекрестную энтропию простых, но наглядных примеров. Например, с помощью следующего кода можно создать гипотетическую унитарно закодированную истинную метку и гипотетический вектор `probs`, а также вычислить соответствующее значение категориальной перекрестной энтропии:

```
const oneHotTruth = tf.tensor1d([0, 1, 0]);
const probs = tf.tensor1d([0.2, 0.5, 0.3]);
tf.metrics.categoricalCrossentropy(oneHotTruth, probs).print();
```

Результат будет равен примерно 0,693. Это значит: если вероятность, присвоенная моделью фактическому классу, равна 0,5, значение `categoricalCrossentropy` равно 0,693. Можете проверить это с помощью псевдокода из листинга 3.10. Попробуйте также увеличить или уменьшить значение с 0,5 и посмотреть, как поменяется `categoricalCrossentropy` (см. примеры в табл. 3.5). В табл. 3.5 есть также столбец для MSE между унитарной истинной меткой и вектором `probs`.

Таблица 3.5. Значения категориальной перекрестной энтропии при различных вероятностях на выходе модели. В основе всех примеров (строк), без потери общности, лежит сценарий с тремя классами (как в случае набора данных «Ирисы Фишера»), причем фактический класс примеров — второй из них

Унитарная истинная метка	Probs (выходной сигнал многомерной логистической функции)	Бинарная перекрестная энтропия	MSE
[0, 1, 0]	[0,2, 0,5, 0,3]	0,693	0,127
[0, 1, 0]	[0,0, 0,5, 0,5]	0,693	0,167
[0, 1, 0]	[0,0, 0,9, 0,1]	0,105	0,006
[0, 1, 0]	[0,1, 0,9, 0,0]	0,105	0,006
[0, 1, 0]	[0,0, 0,99, 0,01]	0,010	0,00006

Если сравнить строки 1 и 2 или 3 и 4 этой таблицы, становится ясно, что изменение элементов `probs`, не относящихся к фактическому классу примера, никак не влияет на категориальную перекрестную энтропию, хотя может менять значение MSE между унитарной истинной меткой и `probs`. Кроме того, как и в случае бинарной перекрестной энтропии, MSE меняется все медленнее по мере приближения значения `probs` для фактического класса к 1, а потому хуже подходит для стимуляции роста вероятности правильного класса, чем категориальная перекрестная энтропия. Именно по этим причинам категориальная перекрестная энтропия лучше подходит для роли функции потерь, чем MSE для задач многоклассовой классификации.

3.3.4. Матрица различий: детальный анализ многоклассовой классификации

Если нажать кнопку Train Model from Scratch (Обучить модель с нуля) на веб-странице примера, можно за несколько секунд получить обученную модель. Как демонстрирует рис. 3.9, безошибочность модели уже после 40 эпох обучения практически идеальна: набор данных «Ирисы Фишера» маленький и границы между классами в пространстве признаков четко очерчены.

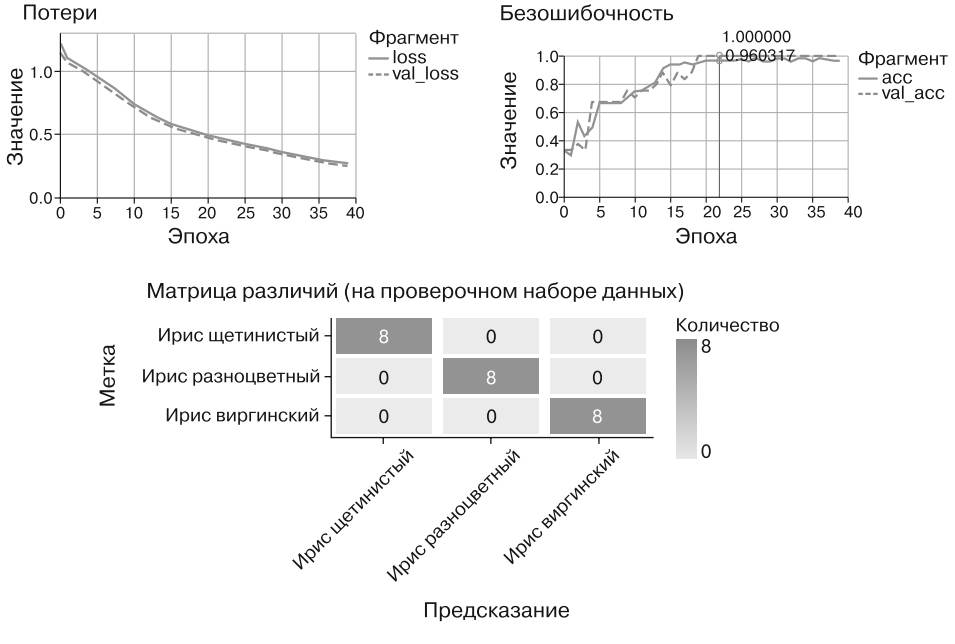


Рис. 3.9. Типичный результат обучения модели iris в течение 40 эпох. *Слева вверху:* функция потерь в зависимости от эпохи обучения. *Справа вверху:* безошибочность в зависимости от эпохи обучения. *Внизу:* матрица различий

Внизу рис. 3.9 показано еще одно средство для описания поведения многоклассового классификатора — *матрица различий* (confusion matrix). Она делит результаты работы многоклассового классификатора по фактическим классам примеров и классам, предсказанным моделью. Это квадратная матрица формы `[numClasses, numClasses]`. Элемент на позиции `[i, j]` (строка `i` и столбец `j`) равен количеству примеров, фактически относящихся к классу `i`, которые модель отнесла к классу `j`. Следовательно, диагональные элементы матрицы различий отражают количество верно классифицированных примеров данных. Матрица различий идеального многоклассового классификатора не должна содержать ненулевых внедиагональных элементов, подобно приведенной на рис. 3.9.

Помимо итоговой матрицы различий, пример с набором данных «Ирисы Фишера» выводит матрицу различий в конце каждой эпохи обучения с помощью обратного вызова `onTrainEnd()`.

На первых эпохах матрица различий не такая идеальная, как на рис. 3.9. Как видно на рис. 3.10, 5 из 24 входных примеров были классифицированы неправильно, соответственно, безошибочность составляет 79 %. Впрочем, информации в этой матрице намного больше, чем одно число: она показывает, в каких классах встречается больше ошибок, а в каких — меньше. В этом конкретном примере пять из восьми цветков второго класса классифицированы неправильно (отнесены к третьему классу), в то время как остальные классифицированы правильно. Отсюда видно, что при многоклассовой классификации матрица различий несет больше информации, чем просто метрика безошибочности, подобно тому как точность и полнота вместе лучше иллюстрируют ситуацию при бинарной классификации, чем безошибочность. Матрица различий содержит информацию, полезную для принятия связанных с моделью и процессом обучения решений. Например, некоторые виды ошибок приводят к более серьезным последствиям, чем другие. Скажем, перепутать спортивный сайт с игровым не так страшно, как перепутать спортивный с фишинговым. В подобных случаях можно подстроить гиперпараметры модели, чтобы минимизировать число наиболее серьезных ошибок.

Рис. 3.10. Пример «неидеальной» матрицы различий с ненулевыми внедиагональными элементами. Эта матрица была сгенерирована всего после двух эпох обучения, до того как обучение сошлось

На вход всех приведенных выше моделей подается массив чисел. Другими словами, каждый входной пример представляется в виде простого списка чисел фиксированной длины, причем упорядоченность элементов неважна, лишь бы она была одинаковой для всех примеров данных. И хотя подобный тип моделей охватывает значительное подмножество важных на практике задач машинного обучения, он далеко не единственный. В следующих главах мы рассмотрим более сложные входные типы данных, включая изображения и последовательности. В главе 4 начнем обсуждение с изображений — распространенного и весьма полезного типа входных данных, для работы с которыми были созданы специальные виды нейронных сетей, позволяющие добиться поистине нечеловеческой степени безошибочности.

Упражнения

1. При создании нейронных сетей для задачи предсказания цен на бостонскую недвижимость мы остановились на модели с двумя скрытыми слоями. С учетом сказанного выше о том, как суперпозиция нелинейных функций позволяет расширить разрешающие возможности моделей, повысит ли добавление в модель дополнительных скрытых слоев безошибочность оценок? Для проверки модифицируйте файл `index.js` и снова запустите обучение и оценку.
   А. Почему добавление дополнительных скрытых слоев не повышает безошибочность оценки?
   Б. Откуда такой вывод? (Подсказка: обратите внимание на погрешность на обучающем наборе данных.)
2. Посмотрите, как код в листинге 3.6 вычисляет и отрисовывает кривую ROC в начале каждой эпохи обучения с помощью обратного вызова `onEpochBegin`. Следуя этому шаблону, попробуйте изменить тело функции обратного вызова так, чтобы выводить значения точности и полноты (вычисленные на контрольном наборе данных) в начале каждой эпохи. Опишите, как они меняются по мере обучения.
3. Изучите код в листинге 3.7 и выясните, как он вычисляет кривую ROC. Следуя этому примеру, напишите новую функцию `drawPrecisionRecallCurve()`, которая, как ясно из названия, вычисляет и визуализирует кривую «точность/полнота». А когда напишете, вызовите ее из обратного вызова `onEpochBegin`, чтобы отрисовывать кривую «точность/полнота» рядом с кривой ROC в начале каждой эпохи обучения. Возможно, вам придется внести определенные изменения или дополнения в файл `ui.js`.
4. Пусть вам известны FPR и TPR результатов работы бинарного классификатора. Можете ли вы вычислить общую безошибочность модели на основе этих двух чисел? Если нет, то какая еще информация вам требуется?
5. Определение как бинарной перекрестной энтропии (см. подраздел 3.2.4), так и категориальной перекрестной энтропии (см. подраздел 3.3.3) основаны на понятии натурального логарифма (логарифма по основанию e). Что будет, если в их опре-

делениях натуральный логарифм заменить логарифмом по основанию 10? Как это повлияет на обучение и вывод бинарного и многоклассового классификаторов?

6. На основе псевдокода из листинга 3.4 напишите настоящий код на JavaScript для поиска гиперпараметров по сетке и воспользуйтесь им для оптимизации гиперпараметров для двухслойной модели поиска цен на бостонскую недвижимость в листинге 3.1. А именно, подберите количество нейронов скрытого слоя и скорость обучения. Не бойтесь экспериментировать с диапазонами для поиска количества нейронов и скорости обучения. Обратите внимание, что специалисты по машинному обучению для такого поиска обычно используют идущие приблизительно в геометрической (логарифмической) прогрессии интервалы (например, количество нейронов = 2, 5, 10, 20, 50, 100, 200...).

Резюме

- Задачи классификации отличаются от задач регрессии необходимостью дискретных предсказаний.
- Существует две разновидности классификации: бинарная и многоклассовая. При бинарной классификации заданный входной пример может относиться к одному из двух классов, а при многоклассовой — к трем или более.
- Бинарную классификацию обычно можно рассматривать как задачу обнаружения событий определенного типа или важных для нас объектов — называемых позитивными — среди всех входных примеров данных. Благодаря этому для количественного выражения различных аспектов поведения бинарного классификатора, помимо безошибочности, можно использовать такие метрики, как точность, полнота и FPR.
- В задачах бинарной классификации зачастую приходится соблюдать баланс между необходимостью найти все позитивные примеры данных и минимизировать число ложнопозитивных результатов (ошибочных срабатываний). Для количественного выражения и визуализации этого соотношения удобна кривая ROC вместе с сопутствующей ей метрикой AUC.
- Последний (выходной) слой нейронной сети для бинарной классификации должен включать сигма-функцию активации, а в качестве функции потерь во время обучения следует применять бинарную перекрестную энтропию.
- В предназначенных для многоклассовой классификации нейронных сетях выходной целевой признак обычно кодируется с помощью унитарного представления. Они должны использовать многомерную логистическую функцию активации в выходном слое, а роль функции потерь при обучении должна играть категориальная перекрестная энтропия.
- При многоклассовой классификации матрицы различий позволяют получить более подробную информацию о допускаемых моделью ошибках, чем дает метрика безошибочности.

- В табл. 3.6 приведен обзор методик, рекомендуемых для работы с наиболее распространенными типами задач машинного обучения, описанными выше (регрессия, бинарная классификация и многоклассовая классификация).
- Гиперпараметры представляют собой параметры структуры модели машинного обучения, свойств ее слоев и процесса обучения. В отличие от весовых параметров модели, они: 1) не меняются в процессе обучения модели и 2) обычно дискретны. Оптимизация гиперпараметров — процесс поиска значений гиперпараметров, при которых достигается минимум функции потерь на проверочном наборе данных. Оптимизация гиперпараметров все еще активно исследуется. В настоящее время для нее чаще всего применяются методы поиска по сетке, случайного поиска и байесовские методы.

Таблица 3.6. Обзор наиболее распространенных типов задач машинного обучения, соответствующих функций активации последнего слоя и функций потерь, а также метрик, удобных для количественного выражения качества работы модели

Тип задачи	**Функция активации выходного слоя**	**Функция потерь**	**Соответствующие метрики, поддерживаемые при вызовах Model.fit()**	**Дополнительные метрики**
Регрессия	'linear' (по умолчанию)	'meanSquaredError' или 'meanAbsoluteError'	(Аналогично функции потерь)	
Бинарная классификация	'sigmoid'	'binaryCrossentropy'	'accuracy'	Точность, полнота, кривая «точность/полнота», кривая ROC, AUC
Однозначная многоклассовая классификация	'softmax'	'categoricalCrossentropy'	'accuracy'	Матрица различий

4 Распознавание изображений и звуковых сигналов с помощью сверточных сетей

В этой главе

- Представление изображений и различной сенсорной информации (например, звука) в виде многомерных тензоров.
- Что такое сверточные нейронные сети, как они работают и почему особенно хорошо подходят для задач машинного обучения, связанных с изображениями.
- Как написать и обучить сверточную нейронную сеть на TensorFlow.js для классификации рукописных цифр.
- Ускорение обучения моделей с помощью Node.js.
- Применение сверточных нейронных сетей к аудиоданным для распознавания устной речи.

Нынешняя революция в сфере глубокого обучения началась с прорывов в решении задач распознавания изображений, например конкурса ImageNet. С изображениями связан широкий диапазон полезных и интересных в техническом отношении задач: от распознавания их содержимого до разбиения на осмысленные составные части, от определения местонахождения объектов в изображениях до синтезирования изображений. Этот подраздел машинного обучения иногда называют *машинным зрением* (computer vision)[1]. Методики машинного зрения часто применяются в областях,

[1] Учтите, что машинное зрение — обширная сфера знаний, частично связанная с не относящимися к машинному обучению методиками, выходящими за рамки этой книги.

совершенно не связанных со зрением или изображениями (например, при обработке текстов на естественном языке), — еще один довод в пользу исследования возможностей применения глубокого обучения в машинном зрении[1]. Но прежде, чем углубиться в задачи машинного зрения, обсудим способы представления изображений в глубоком обучении.

4.1. От векторов к тензорам: представление изображений

В предыдущих двух главах мы рассмотрели задачи машинного обучения, в том числе использование численных входных данных. Например, алгоритм предсказания времени скачивания из главы 2 принимает в качестве входных данных одно число (размер файла). Роль входных данных в задаче предсказания цен на бостонскую недвижимость играет массив из 12 чисел (число комнат, уровень преступности и т. д.). Объединяет эти задачи тот факт, что каждый из входных примеров можно представить в виде «плоского» (без дополнительных уровней вложенности) массива чисел, соответствующего одномерному тензору в TensorFlow.js. Изображения в глубоком обучении представляются иначе.

Для представления изображения мы будем использовать трехмерный тензор. Первые два измерения этого тензора — обычные высота и ширина. Третье — цветовой канал. Например, цвета часто кодируются с помощью схемы RGB. В этом случае получается три канала, по одному для каждого из трех цветов, в результате чего размер третьего измерения равен 3. Кодированное с помощью RGB цветное изображение размером 224 × 224 пиксела можно представить в виде трехмерного тензора формы `[224, 224, 3]`. В некоторых задачах машинного зрения есть возможность обрабатывать не цветные (например, в оттенках серого) изображения. В подобных случаях число каналов равно 1 — при представлении в виде трехмерного тензора это приводит к тензору формы `[height, width, 1]` (см. пример на рис. 4.1)[2].

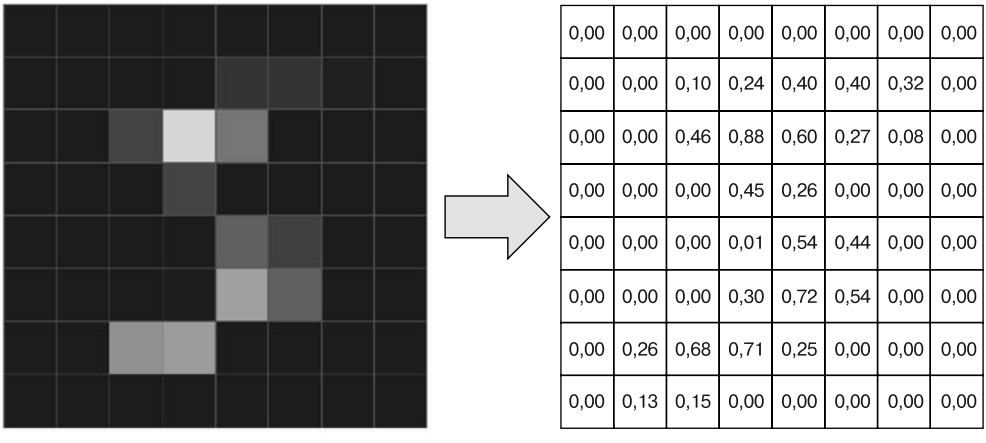
Подобный режим кодирования изображений называется *«высота — ширина — канал»* (height-width-channel, *HWC*). При обработке изображений методами глубокого обучения наборы изображений обычно объединяются в батчи — так можно эффективнее распараллеливать вычисления. При обработке изображений по батчам измерение отдельных изображений — всегда первое действие, аналогично тому, как мы объединяли одномерные тензоры в двумерный тензор батчей в главах 2 и 3. Следовательно, батч изображений представляет собой четырехмерный тензор с такими

---

[1] Читателям, которых особенно интересуют вопросы применения глубокого обучения в машинном зрении, рекомендуем обратиться к книге: *Elgendy M.* Grokking Deep Learning for Computer Vision. — Manning.

[2] В качестве альтернативного представления можно схлопнуть все пикселы изображения и соответствующие им значения цветов в одномерный тензор (плоский числовой массив). Но тогда становится сложнее извлечь пользу из связей между цветовыми каналами пикселов и двумерным пространственным отношением между этими пикселами.

измерениями соответственно, как номер изображения (number, N), высота (height, H), ширина (width, W) и цветовой канал (channel, C). Этот формат называют *NHWC*. Существует и альтернативный формат, с другим порядком четырех измерений, — он называется *NCHW*. Как понятно из названия, в формате NCHW измерение каналов предшествует измерениям высоты и ширины. TensorFlow.js может работать с обоими форматами, но в книге для единообразия мы будем использовать только формат по умолчанию, NHWC.



0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
0,00	0,00	0,10	0,24	0,40	0,40	0,32	0,00
0,00	0,00	0,46	0,88	0,60	0,27	0,08	0,00
0,00	0,00	0,00	0,45	0,26	0,00	0,00	0,00
0,00	0,00	0,00	0,01	0,54	0,44	0,00	0,00
0,00	0,00	0,00	0,30	0,72	0,54	0,00	0,00
0,00	0,26	0,68	0,71	0,25	0,00	0,00	0,00
0,00	0,13	0,15	0,00	0,00	0,00	0,00	0,00

Рис. 4.1. Представление изображения из набора данных MNIST в виде тензоров для глубокого обучения. Ради наглядности мы уменьшили изображение MNIST с 28×28 до 8×8. Это изображение в оттенках серого, в результате чего получается тензор HWC формы [8, 8, 1]. Единственный его цветовой канал по последнему измерению мы опустим на схеме

4.1.1. Набор данных MNIST

В этой главе мы займемся задачей машинного зрения, связанной с набором данных рукописных цифр MNIST[1]. Он настолько важен и часто используется, что его считают своего рода Hello, world машинного зрения и глубокого обучения. Набор данных MNIST меньше и старше, чем большинство наборов данных, встречающихся на практике при глубоком обучении. Тем не менее познакомиться с этим широко

[1] Аббревиатура MNIST расшифровывается как модифицированный NIST (Modified NIST). NIST всего лишь означает, что источником этого набора данных стал Национальный институт стандартов и технологий США (US National Institute of Standards and Technology) (около 1995 года). Определение «модифицированный» относится к следующим изменениям, внесенным в исходный набор данных NIST: 1) нормализация изображений к одному размеру растра — 28×28 со сглаживанием для большей однородности обучающего и контрольного наборов данных и 2) четкое разделение множеств людей, писавших исследуемые цифры для обучающего и контрольного наборов данных. Благодаря этим модификациям с набором данных стало проще работать и он лучше подходит для объективной оценки степени безошибочности модели.

используемым примером отнюдь не помешает, ведь именно на нем обычно в первую очередь тестируют новые алгоритмы глубокого обучения.

Все примеры данных в MNIST представляют собой изображения в оттенках серого, имеющие размер 28 × 28 (см. пример на рис. 4.1). Они были получены из реальных написанных вручную цифр от 0 до 9. Размер изображения 28 × 28 вполне достаточен для уверенного распознавания этих простых фигур, хотя он меньше размеров изображений, обычно используемых в задачах машинного зрения. Каждому изображению соответствует метка, указывающая, какая из десяти цифр на нем в самом деле представлена. Подобно наборам данных времени скачивания и цен на бостонскую недвижимость, данные разбиты на обучающий и контрольный наборы. Обучающий набор данных состоит из 60 000 изображений, а контрольный содержит 10 000 изображений. Набор данных MNIST[1] более или менее сбалансирован в том смысле, что к каждой из десяти категорий (десяти цифр) относится примерно одинаковое количество изображений.

4.2. Ваша первая сверточная нейронная сеть

Исходя из представления данных изображений и их меток нам известно, какие входные данные должна получать нейронная сеть, предназначенная для работы с набором MNIST, и какой выходной сигнал она должна генерировать. Входной сигнал этой сети представляет собой тензор формата NHWC формы `[null, 28, 28, 1]`. Выходной сигнал представляет собой тензор формы `[null, 10]`, где второе измерение соответствует десяти возможным цифрам. Это каноническое унитарное кодирование целевых переменных задачи многоклассовой классификации. Оно аналогично унитарному кодированию видов ирисов из примера в главе 3. Теперь можно заняться непосредственно реализацией сверточной нейронной сети — метода, рекомендуемого для использования в задачах классификации изображений наподобие MNIST. Не пугайтесь слова «сверточный», оно лишь обозначает определенную математическую операцию, о которой мы подробнее поговорим позже.

Код вы можете найти в каталоге `mnist` репозитория tfjs-examples. Как и в предыдущих примерах, получить и запустить код можно с помощью следующих команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/website-phishing
yarn && yarn watch
```

Листинг 4.1 представляет собой выдержку из основного файла кода `index.js` примера MNIST — функцию, создающую сверточную сеть, с помощью которой мы собираемся решать задачу MNIST.

---

[1] См.: *LeCun Y., Cortes C., Burges C. J. C.* The MNIST Database of Handwritten Digits. http://yann.lecun.com/exdb/mnist/.

Листинг 4.1. Описание сверточной модели для набора данных MNIST

```
function createConvModel() {
  const model = tf.sequential();

  model.add(tf.layers.conv2d({                          // Первый слой conv2d
    inputShape: [IMAGE_H, IMAGE_W, 1],
    kernelSize: 3,
    filters: 16,
    activation: 'relu'
  }));
  model.add(tf.layers.maxPooling2d({                    // Субдискретизация с выбором
    poolSize: 2,                                        // максимального значения после свертки
      strides: 2
  }));

  model.add(tf.layers.conv2d({                          // Повтор «лейтмотива»
    kernelSize: 3, filters: 32, activation: 'relu'}));  // conv2d-maxPooling2d
  model.add(tf.layers.maxPooling2d({poolSize: 2, strides: 2}));

  model.add(tf.layers.flatten());                       // Схлопывание тензора — подготовка его для плотных слоев
  model.add(tf.layers.dense({
    units: 64,
    activation:'relu'                                   // Для решения задачи многоклассовой классификации
  }));                                                  // используется многомерная логистическая функция активации
  model.add(tf.layers.dense({units: 10, activation: 'softmax'}));

  model.summary();                                      // Выводим текстовую сводку топологии модели
  return model;
}
```

Последовательная модель, формируемая кодом из листинга 4.1, состоит из семи слоев, создаваемых по очереди с помощью вызовов метода `add()`. Прежде чем изучать подробности операций, выполняемых слоями, взглянем на общую архитектуру модели (рис. 4.2). Как демонстрирует схема, первые пять слоев модели содержат повторяющийся паттерн из групп слоев conv2d-maxPooling2d, за которыми следует схлопнутый слой. Именно в этих группах слоев conv2d-maxPooling2d заключается основной механизм выделения признаков. Каждый из этих слоев преобразует входное изображение в выходное. В основе работы слоя conv2d лежит *сверточное ядро* (convolutional kernel), «скользящее» по измерениям высоты и ширины входного изображения. При этом на каждой позиции оно умножается на входные пикселы, и полученные произведения суммируются и пропускаются через нелинейность. В результате получается пиксел выходного изображения. Слои maxPooling2d работают аналогично, но без ядра. Пропуская входные данные изображений через последовательные слои свертки и субдискретизации, мы получаем все меньшие по размеру и все более абстрактные в пространстве признаков тензоры. Выходной сигнал последнего слоя субдискретизации преобразуется в одномерный тензор с помощью схлопывания. Этот схлопнутый одномерный тензор затем попадает в плотный слой (на схеме не показан).

Сверточную сеть можно рассматривать как MLP, с предварительной обработкой в виде свертки и субдискретизации. Именно с таким MLP мы имели дело в задачах

предсказания цен на бостонскую недвижимость и обнаружения фишинговых сайтов: он состоит просто из плотных слоев с нелинейными функциями активации. Приведенная здесь сверточная нейронная сеть отличается тем, что входным сигналом для MLP служит выходной сигнал каскада слоев conv2d и maxPooling2d. Эти слои специально предназначены для выделения полезных признаков из входных сигналов в виде изображений. Подобная архитектура — результат многолетних исследований в области нейронных сетей: она обеспечивает намного более высокие показатели безошибочности, чем подача значений пикселов изображений непосредственно в MLP.

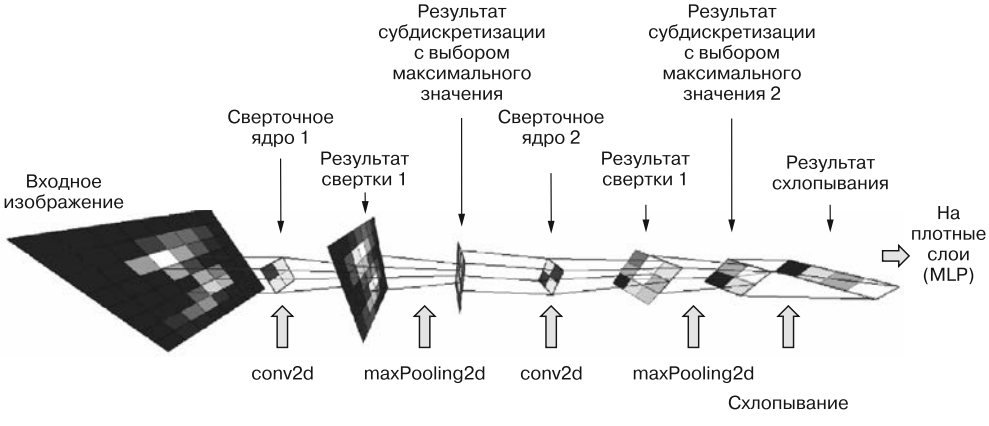


Рис. 4.2. Укрупненная архитектура простой сверточной сети, подобной показанной в листинге 4.1. Ради наглядности размеры изображений и промежуточных тензоров меньше, чем в настоящей модели, описанной в листинге 4.1. Равно как и размеры сверточных ядер. Обратите внимание, что на этой схеме показано по одному каналу в каждом из промежуточных четырехмерных тензоров, в то время как в самой модели у промежуточных тензоров несколько каналов

После этого общего обзора сверточной сети для задачи MNIST можем перейти к внутреннему устройству слоев нашей модели.

4.2.1. Слой conv2d

Первый слой — слой conv2d, выполняющий двумерную свертку. Это первый сверточный слой в книге. Что он делает? Conv2d преобразует изображение в изображение, а именно, преобразует четырехмерный (NHWC) тензор в другой четырехмерный тензор изображения, возможно, с другими высотой, шириной и количеством каналов. (Может показаться странным, что conv2d работает с четырехмерными тензорами, но не забудьте, что в них есть два дополнительных измерения: одно для батчей, а второе — для каналов.) Интуитивно его можно считать неким аналогом группы простых фильтров Photoshop[1], создающим эффекты наподобие размытия

[1] Этой аналогией мы обязаны докладу Аши Кришнан «Глубокое обучение на JS» (*Krishnan A.* Deep Learning in JS) на конференции JSConf EU 2018: http://mng.bz/VPa0.

и наведения на резкость. Эти эффекты достигаются с помощью двумерной свертки со «скольжением» небольшого «окна» пикселов (*сверточного ядра*, либо просто *ядра*) по входному изображению. На каждой позиции в ходе этого «скольжения» ядро попиксельно умножается на перекрываемый им небольшой участок входного изображения. А затем эти попиксельные произведения суммируются, в результате чего получаются пикселы выходного изображения.

Количество параметров слоя conv2d больше, чем у плотного слоя. Два главных параметра слоя conv2d — `kernelSize` и `filters`. Чтобы понять их суть, сначала придется в общих чертах описать, как происходит двумерная свертка.

На рис. 4.3 приведена подробная схема работы двумерной свертки. Мы предполагаем, что тензор входного изображения *(слева вверху)* состоит из простого примера данных, чтобы было удобно изображать его на бумаге. Мы также используем следующие параметры операции conv2d: `kernelSize = 3` и `filters = 3`. Ядро представляет собой тензор формы `[3, 3, 2, 3]`, поскольку цветовых каналов здесь два (несколько необычный вариант ради удобства иллюстрации), что меньше типичного значения `3` или `4` (например, в случае RGB или RGBA). Первые два числа (`3` и `3`) отражают высоту и ширину ядра, определяемые параметром `kernelSize`. Третье измерение (`2`) — число входных каналов. А что же четвертое измерение (`3`)? Это количество фильтров, соответствующее последнему измерению выходного тензора операции conv2d.

Итак, пусть свойство `filters` (число фильтров) слоя conv2d равно `3`, `kernelSize` — `[3, 3]`, `strides` — `[1, 1]`, тогда первым шагом двумерной свертки будет «скольжение» по измерениям высоты и ширины с выделением маленьких фрагментов исходного изображения — высоты `3` и ширины `3` — в соответствии со значением `filterSize` слоя. Глубина также совпадает с исходным изображением. На втором шаге вычисляются скалярные произведения каждого из фрагментов размером $3 \times 3 \times 2$ и сверточного ядра (то есть выполняется фильтрация). Подробности этих операций скалярного произведения приведены на рис. 4.4. Ядро представляет собой четырехмерный тензор и состоит из трех фильтров. Скалярное произведение фрагмента изображения и фильтра вычисляется отдельно для каждого из трех фильтров. Фрагмент изображения умножается на фильтр попиксельно, после чего произведения суммируются и получается пиксел выходного тензора. А поскольку ядро содержит три фильтра, то каждый из фрагментов изображения преобразуется в «стопку» из трех пикселов. Данная операция скалярного произведения проводится над всеми фрагментами изображения, а полученные «стопки» из трех пикселов объединяются в выходной тензор, в данном случае формы `[2, 3, 3]`.

Если рассматривать выходной сигнал как тензор изображения (что вполне логично!), то фильтры можно считать числом каналов выходного сигнала. В отличие от входного изображения каналы выходного тензора никакого отношения к цветам не имеют, а представляют различные визуальные признаки входного изображения, усвоенные моделью из обучающих данных. Например, один фильтр может быть чувствителен к проходящим под определенным углом прямолинейным границам между темными и светлыми участками изображения, а другой — к углам коричневого цвета и т. д. Мы вернемся к этому вопросу позже.

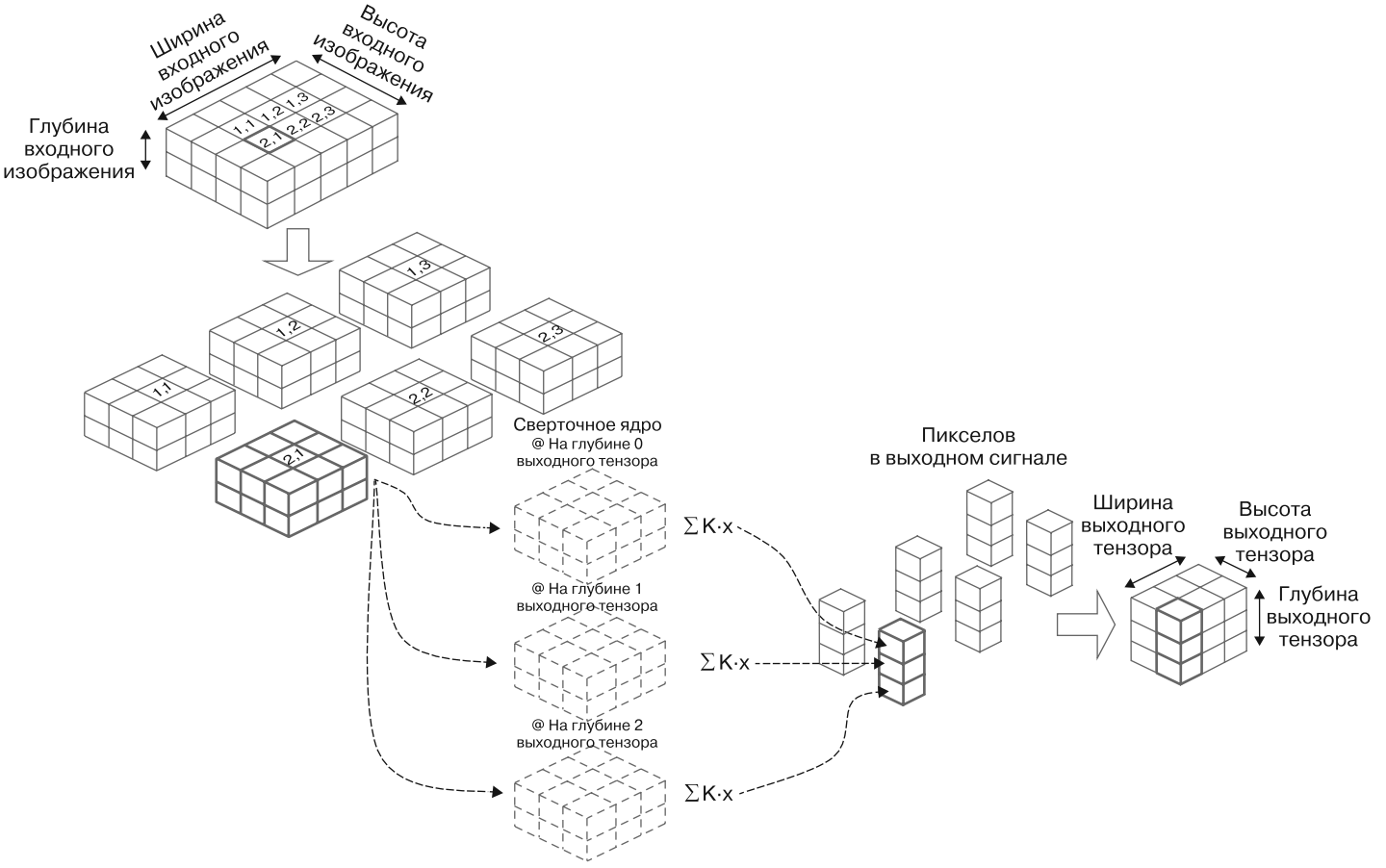


Рис. 4.3. Функционирование слоя conv2d на примере. Для простоты предполагаем, что входной тензор *(слева вверху)* содержит только одно изображение, а потому является трехмерным. Его измерения: высота, ширина и глубина (цветовой канал). Измерение батчей для простоты опущено. Глубина тензора входного изображения взята равной 2

Упомянутое выше «скольжение» представляет собой выделение небольших фрагментов из входного изображения. Ширина и высота всех фрагментов равна `kernelSize` (в данном случае `3`). А поскольку высота входного изображения равна 4, значит, возможны ровно две позиции окна по измерению высоты, ведь окно размером 3×3 не должно выходить за границы изображения. Аналогично ширина (`5`) входного изображения означает три возможные позиции окна по измерению ширины. Следовательно, суммарно выделяется $2 \times 3 = 6$ фрагментов изображения.

На каждой из позиций окна вычисляется скалярное произведение. Напомним, что форма сверточного ядра — `[3, 3, 2, 3]`. Четырехмерный тензор можно разбить по последнему измерению на три отдельных трехмерных тензора формы `[3, 3, 2]`, как показано пунктирными линиями на рис. 4.3. Для получения одного из пикселов выходного тензора фрагмент изображения умножается на один из этих трехмерных тензоров попиксельно, после чего все полученные $3 \times 3 \times 2 = 18$ значений суммируются.

На рис. 4.4 шаг скалярного произведения показан подробнее. Одинаковая форма фрагмента изображения и среза сверточного ядра не совпадение — мы выделяли фрагменты изображения в соответствии с формой ядра! Операция умножения и суммирования повторяется для всех трех срезов ядра, в результате чего мы получаем набор из трех чисел. Затем эта операция скалярного произведения повторяется для оставшихся фрагментов изображения, в результате чего мы получаем шесть столбцов трех кубиков нашего тензора. Эти столбцы объединяются, и получается выходной тензор с формой HWC `[2, 3, 3]`.

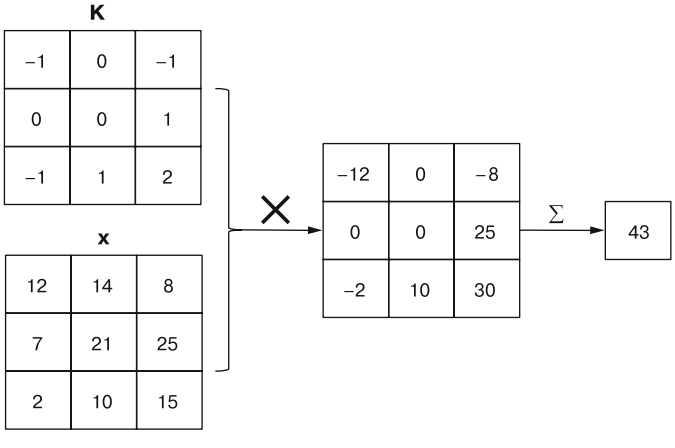


Рис. 4.4. Наглядная иллюстрация операции скалярного произведения (умножения и суммирования) в ходе двумерной свертки — одного из шагов технологического процесса, изображенного на рис. 4.3. Ради наглядности считаем, что фрагмент изображения (x) включает только один цветовой канал. Форма фрагмента изображения — [3, 3, 1], то есть такая же, как и у среза сверточного ядра (K). В результате первого шага — поэлементного умножения — получается еще один тензор формы [3, 3, 1]. Элементы нового тензора суммируются (обозначено символом $\sum$) и получается итоговый результат

Аналогично плотному слою слой conv2d включает член смещения, прибавляемый к результату свертки. Кроме того, в слое conv2d обычно используется нелинейная функция активации. В данном примере ее роль играет ReLU. Напомним, что в пункте «Избегаем нагромождения слоев без нелинейностей» главы 3 на с. 115 мы предостерегали: два идущих подряд плотных слоя без нелинейностей эквивалентны одному плотному слою. То же самое справедливо и для слоев conv2d: два идущих подряд подобных слоя без нелинейной функции активации математически эквивалентны одному слою conv2d с бо́льшим ядром, а значит, составлять подобным образом сверточную нейронную сеть смысла не имеет, желательно этого избегать.

Уф! С нюансами функционирования слоев conv2d покончено. Давайте на минуту остановимся и взглянем, что на самом деле дает нам слой conv2d. По существу, он представляет собой особый способ преобразования входного изображения в выходное. Высота и ширина выходного изображения обычно меньше, чем у входного. Степень уменьшения размеров зависит от параметра `kernelSize`. А число каналов выходного изображения может быть меньшим, таким же или большим, чем у входного, в зависимости от параметра `filters`.

Итак, слой conv2d представляет собой преобразование одного изображения в другое. Две основные особенности преобразования conv2d — локальность и единство параметров.

- *Локальность* (locality) означает, что на значение конкретного пиксела выходного изображения влияет лишь небольшой фрагмент входного изображения, а не все пикселы этого входного изображения. Размер фрагмента определяется параметром `kernelSize`. Именно это отличает conv2d от плотных слоев: в плотном слое все составляющие входного сигнала влияют на все составляющие выходного. Другими словами, в плотном слое входные и выходные элементы «плотно связаны» (отсюда и его название). Таким образом, можно говорить о «разреженных связях» в слое conv2d. Плотные слои усваивают глобальные закономерности входных данных, а сверточные слои — локальные, то есть проявляющиеся в пределах маленького окна ядра.
- *Единство параметров* (parameter sharing) означает, что на выходной пиксел А его маленький входной фрагмент изображения влияет точно так же, как и на выходной пиксел Б — его входной фрагмент. Дело в том, что при скалярном произведении во всех позициях окна используется одно и то же сверточное ядро (см. рис. 4.3).

Благодаря локальности и единству параметров слой conv2d является высокоэффективным преобразованием одного изображения в другое, если говорить о количестве требуемых параметров. В частности, размер сверточного ядра не меняется с изменением высоты или ширины входного изображения. Возвращаясь к нашему первому слою conv2d из листинга 4.1, видим у него форму ядра `[kernelSize, kernelSize, 1, filter]` (то есть `[5, 5, 1, 8]`), что дает суммарно $5 \times 5 \times 1 \times 8 = 200$ параметров, вне зависимости от размера входных изображений MNIST (28×28 или намного больше). Форма выходного сигнала этого слоя conv2d — `[24, 24, 8]` (не считая измерения батчей). Таким образом, этот слой conv2d преобразует тензор,

состоящий из $28 \times 28 \times 1 = 784$ элементов, в другой тензор, состоящий из $28 \times 28 \times 8 = 4608$ элементов. А сколько бы потребовалось параметров при реализации такого преобразования с помощью плотного слоя? Ответ: $784 \times 4608 = 3\,612\,672$ (не считая смещения), то есть почти в 18 тысяч раз больше, чем у слоя conv2d! Этот мысленный эксперимент наглядно демонстрирует эффективность сверточных слоев.

Прелесть локальности и единства параметров слоя conv2d не только в эффективности, но и в имитации (хотя и приближенной) работы биологических систем зрения. Представьте себе нейроны в сетчатке глаза. На каждый из них воздействует лишь небольшой участок поля зрения глаза — так называемое *рецептивное поле* (receptive field). Расположенные в различных частях сетчатки нейроны реагируют на световые паттерны в своих рецептивных полях примерно одинаково, аналогично единству параметров в слоях conv2d. Более того, слои conv2d, оказывается, отлично подходят для решения задач машинного зрения, как мы скоро увидим в примере MNIST. conv2d — прекрасный слой нейронной сети, обладающий всеми достоинствами: эффективностью, безошибочностью и «биологичностью». Неудивительно, что он столь широко применяется в глубоком обучении.

4.2.2. Слой maxPooling2d

Теперь рассмотрим следующий слой нашей модели — maxPooling2d. Подобно слою conv2d, maxPooling2d представляет собой разновидность преобразования одного изображения в другое. Но преобразование maxPooling2d проще, чем conv2d. Как демонстрирует рис. 4.5, при нем вычисляются максимальные значения пикселов в маленьких фрагментах изображения, далее используемые в качестве значений пикселов выходного изображения. Код описания и добавления в модель слоя maxPooling2d:

```
model.add(tf.layers.maxPooling2d({poolSize: 2,
strides: 2}));
```

В данном случае высота и ширина фрагментов изображения равна 2×2 в силу равного `[2, 2]` значения параметра `poolSize`. Эти фрагменты изображения извлекаются через каждые два пиксела по обоим измерениям. Такой промежуток между фрагментами изображения обуславливается используемым значением `[2, 2]` параметра `strides`. В результате высота и ширина выходного изображения с формой `[12, 12, 8]` в формате HWC составляют лишь половину от высоты и ширины входного изображения (с формой `[24, 24, 8]`), а количество каналов совпадает.

Слой maxPooling2d в сверточной сети служит двум целям. Во-первых, снижает чувствительность сети к точному местоположению ключевых признаков во входном

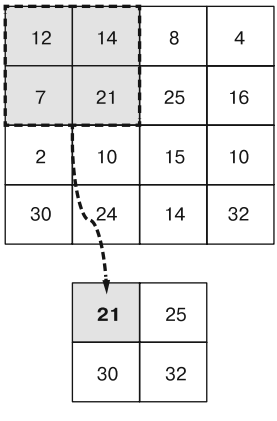


Рис. 4.5. Работа слоя maxPooling2d, на примере крошечного изображения 4×4 с параметрами poolSize = [2, 2] и strides = [2, 2]. Измерение глубины не показано, но операция субдискретизации с выбором максимального значения происходит по отдельности по измерениям

изображении. Например, сеть должна распознавать цифру 8 независимо от того, была ли она сдвинута влево/вправо, вверх/вниз от центра входного изображения 28 × 28. Это свойство называется *независимостью от конкретной позиции* (positional invariance). Чтобы уяснить себе, почему слой maxPooling2d снижает зависимость от конкретной позиции, достаточно осознать: для него неважно, где находится наиболее яркий пиксел в каждом из обрабатываемых фрагментов изображения, лишь бы он не выходил за пределы фрагмента. Конечно, отдельный слой maxPooling2d делает сеть лишь ограниченно нечувствительной к сдвигам в силу ограниченности его окна субдискретизации. Однако совместное использование в одной сети нескольких слоев maxPooling2d позволяет добиться существенно большей независимости от конкретной позиции. Именно такова наша модель MNIST — как и абсолютное большинство используемых на практике сверточных нейронных сетей, — включающая два слоя maxPooling2d.

В качестве мысленного эксперимента представьте, что получится, если разместить непосредственно один над другим два слоя conv2d (назовем их conv2d_1 и conv2d_2) без промежуточного слоя maxPooling2d. Пусть `kernelSize` у каждого из них равен `3`, тогда любой пиксел выходного тензора conv2d_2 является функцией области 5 × 5 входного тензора слоя conv2d_1, то есть размер рецептивного поля у каждого «нейрона» слоя conv2d_2 равен 5 × 5. А что будет, если вставить между этими двумя слоями conv2d (как в нашей сверточной нейронной сети для MNIST) промежуточный слой maxPooling2d? Рецептивное поле нейронов conv2d_2 становится больше: 11 × 11, разумеется, вследствие субдискретизации. Большое количество слоев maxPooling2d в сверточной нейронной сети обеспечивает наличие широких рецептивных полей и независимость от конкретной позиции для слоев на высших уровнях сети. Проще говоря, их поле зрения расширяется!

Во-вторых, слой maxPooling2d уменьшает размеры измерений высоты и ширины входного тензора, существенно снижая объем вычислений, необходимых в последующих слоях и сети в целом. Например, форма выходного тензора первого слоя conv2d `[26, 26, 16]`. После прохождения через слой maxPooling2d тензор приобретает форму `[13, 13, 16]`, то есть количество его элементов уменьшается в четыре раза. Наша сверточная сеть включает еще один слой maxPooling2d, еще более уменьшая размеры набора весов в последующих слоях, а значит, и число поэлементных математических операций в них.

4.2.3. «Лейтмотивы» свертки и субдискретизации

Мы обсудили первый слой maxPooling2d и можем сосредоточить свое внимание на следующих двух слоях сети, описываемых в листинге 4.1 этими строками кода:

```
model.add(tf.layers.conv2d(
    {kernelSize: 3, filters: 32, activation: 'relu'}));
model.add(tf.layers.maxPooling2d({poolSize: 2, strides: 2}));
```

Эти два слоя в точности повторяют предыдущие два (за исключением большего значения параметра `filters` и отсутствия поля `inputShape` в слое conv2d). Этот почти в точности повторяющийся «лейтмотив», состоящий из сверточного слоя и слоя

субдискретизации, очень часто встречается в сверточных сетях. Он выполняет важнейшую миссию: иерархическое выделение признаков. Для пояснения рассмотрим сверточную сеть, предназначенную для классификации животных на фотографиях. На начальных уровнях нейронной сети фильтры (то есть каналы) в сверточном слое могут кодировать простейшие геометрические признаки, например прямые линии, кривые и углы. Эти признаки далее преобразуются в более сложные, например кошачьи глаз, нос и ухо (рис. 4.6). Фильтры слоя на верхнем уровне сверточной сети могут кодировать наличие на изображении всей кошки. Чем выше уровень, тем выше и абстрактность представления и дальше сами признаки от значений отдельных пикселов. Но именно такие абстрактные признаки и позволяют обеспечить высокую безошибочность решения задачи сверточной сети — например, обнаружить кошку, если она изображена на фотографии. Более того, эти признаки не описываются вручную, а выделяются из данных автоматически, путем обучения с учителем. Это классический пример послойного преобразования представлений того типа, который мы описали в главе 1 как сущность глубокого обучения.

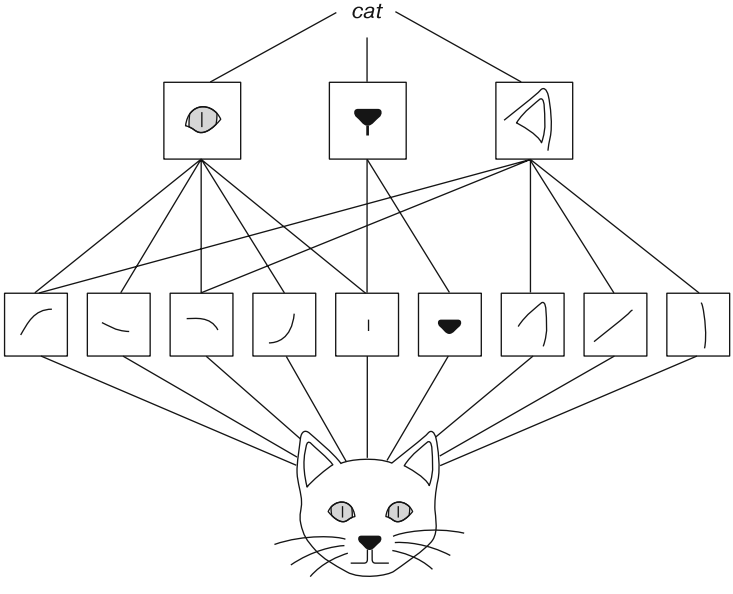


Рис. 4.6. Иерархическое выделение сверточной сетью признаков во входном изображении на примере изображения кошки. Обратите внимание, что входной сигнал сверточной сети приведен внизу, а выходной сигнал — вверху

4.2.4. Слои схлопывания и плотные слои

Пройдя через две группы преобразований conv2d-maxPooling2d, входной тензор становится тензором HWC формы `[4, 4, 16]` (не считая измерения батчей). Следующий слой нашей сверточной сети — слой схлопывания, связывающий предыдущие слои conv2d-maxPooling2d и дальнейшие слои последовательной модели.

Код слоя схлопывания прост, его конструктор не требует параметров:

```
model.add(tf.layers.flatten());
```

Слой схлопывания «сплющивает» многомерный тензор в одномерный с сохранением общего числа элементов. В нашем случае трехмерный тензор формы `[3, 3, 32]` схлопывается в одномерный тензор формы `[288]` (без измерения батчей). Сразу возникает вопрос относительно этой операции «сплющивания»: как упорядочить элементы схлопнутого одномерного тензора, ведь в исходном трехмерном пространстве отсутствует какая-либо внутренняя упорядоченность. Ответ: мы упорядочиваем элементы таким образом, чтобы при обходе по порядку элементов схлопнутого одномерного тензора быстрее всего менялся последний соответствующий исходный индекс (из трехмерного тензора), предпоследний менялся вторым по скорости, а первый индекс менялся медленнее всего. Все это проиллюстрировано на рис. 4.7.

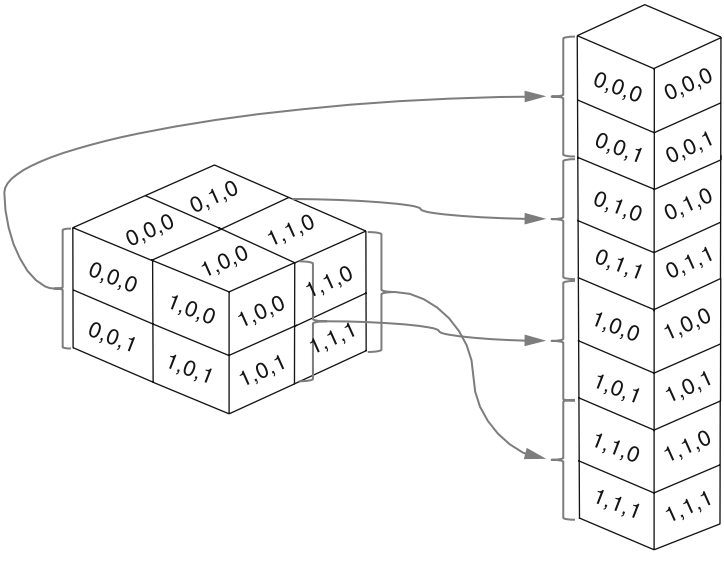


Рис. 4.7. Как работает слой схлопывания. Предполагается, что входной тензор — трехмерный. Ради простоты возьмем небольшой размер всех измерений — 2. Индексы элементов показаны на «лицевой» стороне кубов, представляющих элементы. Слой схлопывания преобразует трехмерный тензор в одномерный с сохранением общего числа элементов. Упорядочение элементов в схлопнутом одномерном тензоре таково, что при обходе по порядку элементов выходного одномерного тензора быстрее всего меняется соответствующее последнее измерение исходного тензора

Какой цели служит слой схлопывания в нашей сверточной сети? Он готовит почву для последующих плотных слоев. Как мы узнали из глав 2 и 3, плотный слой обычно получает одномерный тензор (не считая измерения батчей) в качестве входного сигнала в соответствии с принципами его работы (см. подраздел 2.1.4).

Следующие две строки кода из листинга 4.1 добавляют в нашу сверточную сеть два плотных слоя:

```
model.add(tf.layers.dense({units: 64, activation: 'relu'}));
model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

Почему два плотных слоя, а не один? По той же причине, что и в примере с распознаванием цен на бостонскую недвижимость и примере с обнаружением фишинговых URL из главы 3: чем больше слоев с нелинейной функцией активации, тем выше разрешающие возможности модели. На самом деле эту сверточную нейронную сеть можно считать состоящей из двух моделей, одна поверх другой.

- Модель, включающая слои conv2d, maxPooling2d и слои схлопывания, выделяющие визуальные признаки из входных изображений.
- MLP с двумя плотными слоями, выполняющий классификацию цифр на основе выделенных признаков — именно для этого, по сути, служат два плотных слоя.

В глубоком обучении нередко встречается подобная архитектура, состоящая из выделяющих признаки слоев, за которыми следуют многослойные перцептроны, которые и производят итоговые предсказания. Далее мы рассмотрим еще много аналогичных примеров, начиная от моделей-классификаторов аудиосигналов и до обработки естественного языка.

4.2.5. Обучение сверточной сети

После успешного описания топологии сверточной сети необходимо ее обучить и оценить качество результата обучения. Именно для этого предназначен код в листинге 4.2.

Листинг 4.2. Обучение сверточной сети MNIST и оценка качества ее работы

```
const optimizer = 'rmsprop';
model.compile({
  optimizer,
  loss: 'categoricalCrossentropy',
  metrics: ['accuracy']
});

const batchSize = 320;
const validationSplit = 0.15;
await model.fit(trainData.xs, trainData.labels, {
  batchSize,
  validationSplit,
  epochs: trainEpochs,
  callbacks: {
    onBatchEnd: async (batch, logs) => {
      trainBatchCount++;
      ui.logStatus(
          `Training... (` +
```

Строим график показателя безошибочности и функции потерь во время обучения

```
          `${(trainBatchCount / totalNumBatches * 100).toFixed(1)}%` +
          ` complete). To stop training, refresh or close page.`);
      ui.plotLoss(trainBatchCount, logs.loss, 'train');
      ui.plotAccuracy(trainBatchCount, logs.acc, 'train');
    },
    onEpochEnd: async (epoch, logs) => {
      valAcc = logs.val_acc;
      ui.plotLoss(trainBatchCount, logs.val_loss, 'validation');
      ui.plotAccuracy(trainBatchCount, logs.val_acc, 'validation');
    }
  }
});

const testResult = model.evaluate(
    testData.xs, testData.labels);
```

Оцениваем безошибочность модели на еще не виденных ею данных

Значительная часть приведенного здесь кода посвящена обновлению UI в ходе обучения, например для построения графика изменений значений потерь и безошибочности. Это удобно для мониторинга процесса обучения, но для самого обучения модели значения не имеет. Отметим элементы этого кода, играющие важную роль в самом обучении.

- `trainData.xs` (первый аргумент вызова `model.fit()`) содержит входные изображения MNIST, представленные в виде тензора NHWC формы `[N, 28, 28, 1]`.
- `trainData.labels` (второй аргумент вызова `model.fit()`) содержит входные метки, представленные в виде унитарного кодированного двумерного тензора формы `[N, 10]`.
- Используемая в вызове `model.compile()` функция потерь `categoricalCrossentropy`, хорошо подходящая для таких задач многоклассовой классификации, как MNIST. Напомним, что мы использовали ту же функцию в задаче классификации ирисов в главе 3.
- Заданная в вызове `model.compile()` функция метрики: `'accuracy'`. Она вычисляет, какая часть примеров данных классифицирована правильно, исходя из предсказания на основе наибольшего из десяти элементов выходного сигнала сверточной сети. Опять же это та же метрика, что и ранее, из задачи классификации новостей. Напомним вам, в чем состоит различие между функцией потерь на основе перекрестной энтропии и метрикой безошибочности: первая дифференцируема, что открывает возможность для обучения на основе обратного распространения ошибки, в то время как метрика безошибочности не дифференцируема, но более понятна для человека.
- Задаваемый в вызове `model.fit()` параметр `batchSize`. Вообще говоря, преимущество большего размера батчей состоит в более согласованном и менее подверженном изменениям градиентном обновлении весов модели, по сравнению с батчами меньшего размера. Но чем больше размер батчей, тем больше оперативной памяти требуется при обучении. Учтите также, что при одинаковом размере обучающих данных чем больше размер батчей, тем меньше градиент-

ных обновлений выполняется за одну эпоху. Так что при использовании батчей большого размера не забудьте соответственно увеличить и количество эпох, чтобы не снизить ненароком число обновлений весов за время обучения. Таким образом, должен соблюдаться определенный баланс. В данном случае размер батча относительно невелик — 64, поскольку нам хотелось бы, чтобы этот пример работал на разном аппаратном обеспечении. Как и другие параметры, его можно поменять в исходном коде и обновить страницу, чтобы поэкспериментировать с различными размерами батчей.

- Используемый в вызове `model.fit()` параметр `validationSplit`. Он позволяет нам оставить последние 15 % аргументов `trainData.xs` и `trainData.labels` для проверки в ходе обучения. Как вы уже знаете из предыдущих, не связанных с изображениями моделей, очень важно отслеживать потери и безошибочность на проверочном наборе данных. Это позволяет понять, не *переобучена* ли модель, и, если да, уловить момент, когда она начинает переобучаться. Что такое переобучение (overfitting)? Это такое состояние модели, в котором она слишком сильно обращает внимание на нюансы поступающих в нее во время обучения данных. Настолько сильно, что это отрицательно сказывается на безошибочности ее предсказаний на новых, не виденных ею данных. Далее в этой книге мы посвятим целую главу (см. главу 8) обнаружению переобучения и борьбе с ним.

`model.fit()` — асинхронная функция, так что нам приходится использовать ключевое слово `await`, если последующие действия зависят от того, завершено ли выполнение. Здесь как раз такая ситуация, ведь нам нужно оценить работу модели на контрольном наборе данных после обучения. Эта оценка выполняется с помощью синхронного метода `model.evaluate()`. На вход в `model.evaluate()` подается `testData` с таким же форматом, как вышеупомянутый `trainData`, но с меньшим количеством примеров данных. Модель не видела эти примеры во время вызова `fit()`, а значит, контрольный набор данных не влияет на обучение, а результаты оценки объективно отражают качество модели.

Мы обучаем модель с помощью этого кода в течение десяти эпох (количество указывается в поле для ввода), получая кривые потерь и безошибочности (рис. 4.8). Как демонстрируют графики, функция потерь сходится ближе к концу числа эпох обучения, как и безошибочность. Значения потерь и безошибочности на проверочном наборе данных не слишком отклоняются от соответствующих значений при обучении, а значит, серьезного переобучения в данном случае не наблюдается. Последний вызов `model.evaluate()` возвращает безошибочность, близкую к 99 % (значения, которые получите вы, могут слегка отличаться от запуска к запуску из-за случайности начальных значений весов и неявной «перетасовки» примеров данных случайным образом во время обучения).

Насколько хорошим результатом будет 99 %? С практической точки зрения это вполне сносный результат, но явно не идеальный. При большем числе сверточных слоев можно достичь безошибочности 99,5 %, в то же время повысив число слоев субдискретизации, а также количество фильтров в модели. Впрочем, обучение

подобной сверточной сети в браузере займет намного больше времени, поэтому имеет смысл производить его в среде с менее ограниченными ресурсами, например Node.js. Мы покажем вам, как это сделать, в разделе 4.3.

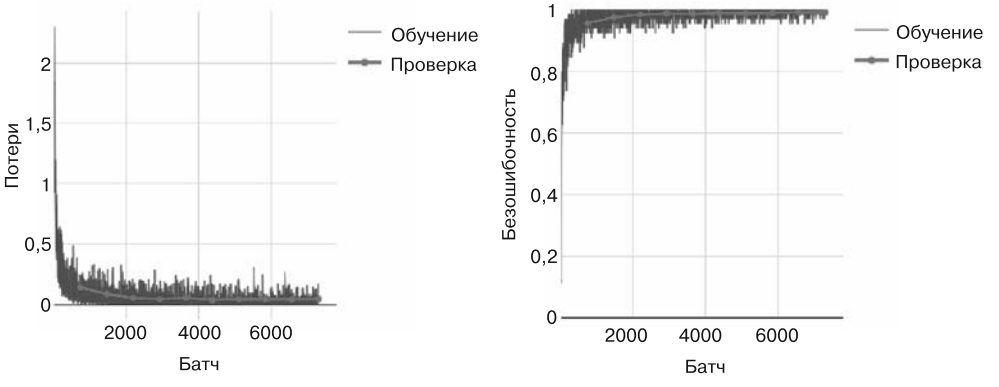


Рис. 4.8. Кривые обучения сверточной сети MNIST. Пройдено десять эпох обучения, примерно по 800 батчей каждая. *Слева:* потери. *Справа:* безошибочность. Значения для обучающего и проверочного наборов данных показаны разными цветами, толщиной линий и метками. Кривые проверочного набора данных содержат меньше точек данных, чем обучающего, поскольку, в отличие от обучения, проверка проводится лишь в конце каждой эпохи

Помните, что MNIST — задача 10-классовой классификации. Так что безошибочность при чисто случайном угадывании равна 10 %, а 99 % — гораздо лучший результат. Но случайное угадывание не слишком высокая планка. Как продемонстрировать пользу слоев conv2d и maxPooling2d модели? Можно ли достичь столь же хорошего результата с помощью одних только старых добрых плотных слоев?

Чтобы ответить на эти вопросы, проведем эксперимент. Код из файла `index.js` включает еще одну функцию для создания модели — `createDenseModel()`. В отличие от функции `createConvModel()` из листинга 4.1, `createDenseModel()` создает последовательную модель, состоящую только из слоев схлопывания и плотных слоев, то есть без использования новых типов слоев, с которыми мы познакомились в этой главе. `createDenseModel()` гарантирует, что общее число параметров в создаваемой ею плотной модели и только что обученной нами сверточной сети примерно одинаково — около 33 000, что делает сравнение более адекватным.

Листинг 4.3. Модель для MNIST из одних слоев схлопывания и плотных слоев для сравнения с нашей сверточной сетью

```
function createDenseModel() {
  const model = tf.sequential();
  model.add(tf.layers.flatten({inputShape: [IMAGE_H, IMAGE_W, 1]}));
  model.add(tf.layers.dense({units: 42, activation: 'relu'}));
  model.add(tf.layers.dense({units: 10, activation: 'softmax'}));
```

```
  model.summary();
  return model;
}
```

Вот текстовая сводка топологии модели, описанной в листинге 4.3:

```
_________________________________________________________________
Layer (type)                 Output shape              Param #
=================================================================
flatten_Flatten1 (Flatten)   [null,784]                0
_________________________________________________________________
dense_Dense1 (Dense)         [null,42]                 32970
_________________________________________________________________
dense_Dense2 (Dense)         [null,10]                 430
=================================================================
Total params: 33400
Trainable params: 33400
Non-trainable params: 0
_________________________________________________________________
```

При тех же параметрах несверточная модель демонстрирует результаты обучения, приведенные на рис. 4.9. После десяти эпох обучения при оценке получаем итоговую безошибочность около 97 %. Может показаться, что разница 2 % совсем незначительна, но с точки зрения частоты появления ошибок несверточная модель оказывается в три раза хуже сверточной. В качестве небольшого упражнения попробуйте увеличить размер несверточной модели, увеличив значение параметра `units` скрытого (первого) плотного слоя в функции `createDenseModel()`. Как вы увидите, даже при большем размере модель с одними плотными слоями не может сравниться в безошибочности со сверточной сетью. Таким образом, благодаря единству параметров и локальности визуальных признаков сверточные сети достигают намного большей безошибочности в задачах машинного зрения при том же или даже меньшем числе параметров, чем несверточные нейронные сети.

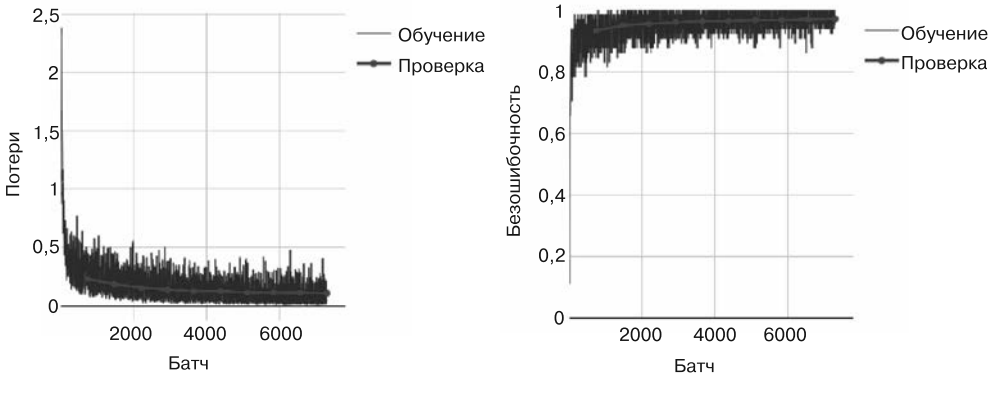


Рис. 4.9. То же самое, что и на рис. 4.8, но при использовании несверточной модели для задачи MNIST, созданной функцией createDenseModel() из листинга 4.3

4.2.6. Предсказания с помощью сверточной сети

Мы получили обученную сверточную сеть. Как же теперь воспользоваться ею для классификации рукописных цифр? Во-первых, необходимо получить доступ к данным изображений. Существует несколько способов сделать это в модели TensorFlow.js. Мы перечислим их и расскажем, в каких случаях они подходят.

Создание тензоров изображений из объектов TypedArray

В некоторых случаях, например, в интересующем нас примере `tfjs-example/mnist`, требуемые данные изображений хранятся в виде объектов `TypedArray` JavaScript. Подробности вы найдете в файле `data.js`, мы не станем рассказывать обо всех нюансах. Заданный `Float32Array` (скажем, содержащийся в переменной `imageDataArray`), представляющий MNIST нужной длины, можно преобразовать в четырехмерный тензор ожидаемой нашей моделью формы[1] так:

```
let x = tf.tensor4d(imageDataArray, [1, 28, 28, 1]);
```

Второй аргумент в вызове `tf.tensor4d()` задает форму создаваемого тензора. Он необходим, поскольку `Float32Array` (и вообще `TypedArray`) представляет собой «плоскую» структуру без какой-либо информации об измерениях изображения. Размер первого измерения — `1`, поскольку в `imageDataArray` всего одно изображение. Как и в предыдущих примерах, модель всегда ожидает наличия измерения батчей при обучении, оценке и выводе, вне зависимости от того, сколько изображений — одно или несколько. Объект `Float32Array`, содержащий батч из нескольких изображений, можно преобразовать в тензор, размер первого измерения которого равен числу изображений:

```
let x = tf.tensor4d(imageDataArray, [numImages, 28, 28, 1]);
```

Функция tf.browser.fromPixels: получение тензоров изображений из HTML-элементов img, canvas и video

Второй способ получить тензоры изображений в браузере — применить функцию `tf.browser.fromPixels()` к HTML-элементам, содержащим данные изображений.

Например, пусть веб-страница содержит элемент `img`, описанный следующим образом:

```
<img id="my-image" src="foo.jpg"></img>
```

Получить данные изображения из элемента `img` можно с помощью такой строки кода:

```
let x = tf.browser.fromPixels(
    document.getElementById('my-image')).asType('float32');
```

[1] См. в приложении Б более подробное руководство по созданию тензоров с помощью низкоуровневого API TensorFlow.js.

Этот код генерирует тензор формы `[height, width, 3]`, где три канала отвечают за кодирование цветов RGB. Вызов `asType` в конце необходим, поскольку метод `tf.browser.fromPixels()` возвращает тензор типа int32, а сверточная сеть ожидает в качестве входных данных тензоры типа float32. Высота и ширина определяются размерами элемента `img`. Если они не соответствуют высоте и ширине, ожидаемым моделью, можно либо поменять атрибуты высоты и ширины элемента `img` (конечно, если это не ухудшает внешний вид UI), либо изменить размер тензора с помощью одного из двух предоставляемых TensorFlow.js соответствующих методов — `tf.image.resizeBilinear()` или `tf.image.resizeNearestNeigbor()`:

```
x = tf.image.resizeBilinear(x, [newHeight, newWidth]);
```

У методов `tf.image.resizeBilinear()` и `tf.image.resizeNearestNeigbor()` одинаковый синтаксис, но они меняют размеры изображений с помощью двух разных алгоритмов. В первом из них значения пикселов нового тензора формируются на основе билинейной интерполяции, а второй выполняет дискретизацию методом ближайших соседей и обычно требует меньше вычислительных ресурсов, чем первый.

Учтите, что тензор, созданный методом `tf.browser.fromPixels()`, не включает измерения батчей. Так что для подачи тензора на вход модели TensorFlow.js его необходимо сначала расширить, добавив измерение батчей, например:

```
x = x.expandDims();
```

Метод `expandDims()` принимает аргумент, определяющий измерение. Но в данном случае его можно опустить, поскольку мы расширяем первое измерение — значение по умолчанию данного аргумента.

Помимо элементов `img`, метод `tf.browser.fromPixels()` работает аналогичным образом и с элементами `canvas` и `video`. Применять метод `tf.browser.fromPixels()` для элементов `canvas` удобно в случаях, когда пользователь может интерактивно менять содержимое холста перед его использованием моделью TensorFlow.js. Например, представьте себе онлайн-приложение для распознавания рукописных цифр или нарисованных от руки фигур. Помимо статических изображений, применяя метод `tf.browser.fromPixels()` к элементам `video`, можно получить покадровые данные изображений с веб-камеры. Именно это происходит в игре Pac-Man, продемонстрированной Нихилем Торатом и Дэниелом Смилковым во время первой презентации TensorFlow.js (см. http://mng.bz/xl0e), в PoseNet[1] и многих других основанных на TensorFlow.js веб-приложениях, использующих веб-камеры. Исходный код Pac-Man вы можете найти на GitHub по адресу http://mng.bz/ANYK.

Как вы видели в предыдущих главах, нужно стараться избегать *асимметрии* (skew) обучающих данных и данных, используемых для вывода. В данном случае наша сверточная сеть MNIST обучается на тензорах изображений, нормализованных к диапазону от 0 до 1. Следовательно, если данные в тензоре `x` входят в другой диапазон, скажем 0–255, необходимо их нормализовать:

```
x = x.div(255);
```

[1] *Oved D.* Real-time Human Pose Estimation in the Browser with TensorFlow.js // Medium, 7 May 2018. http://mng.bz/ZeOO.

При наличии этих данных можно вызывать метод `model.predict()`, чтобы получить предсказания (листинг 4.4).

Листинг 4.4. Вывод с помощью обученной сверточной сети

```
const testExamples = 100;
const examples = data.getTestData(testExamples);

tf.tidy(() => {
  const output = model.predict(examples.xs);
  const axis = 1;
  const labels = Array.from(examples.labels.argMax(axis).dataSync());
  const predictions = Array.from(
      output.argMax(axis).dataSync());

  ui.showTestResults(examples, predictions, labels);
});
```

Используем tf.tidy() для предотвращения утечек памяти WebGL

Получаем класс с наибольшей вероятностью путем вызова argMax()

В коде предполагается, что батч изображений для предсказания содержится в одном тензоре, а именно `examples.xs`. Форма этого тензора — `[100, 28, 28, 1]` (включая измерение батчей), где первое измерение отражает факт наличия 100 изображений, для которых делается предсказание. Метод `model.predict()` возвращает выходной двумерный тензор формы `[100, 10]`. Первое измерение выходного тензора соответствует примерам данных, а второе — десяти возможным цифрам. Каждая строка выходного тензора содержит значения вероятностей каждой из десяти цифр для заданного входного изображения. Чтобы выяснить предсказание, необходимо найти индексы максимальных значений вероятности, изображение за изображением. Эта задача решается в следующих строках кода:

```
const axis = 1;
const labels = Array.from(examples.labels.argMax(axis).dataSync());
```

Функция `argMax()` возвращает индексы максимальных значений по заданной оси координат. В данном случае нужная нам ось — второе измерение, `const axis = 1`. `argMax()` возвращает тензор формы `[100, 1]`. С помощью вызова `dataSync()` мы преобразуем тензор формы `[100, 1]` в объект `Float32Array` длиной 100. Далее `Array.from` преобразует этот `Float32Array` в обычный массив JavaScript, состоящий из 100 целых чисел от 0 до 9. Смысл массива предсказаний совершенно прозрачен: он представляет собой результаты выполненной моделью классификации 100 входных изображений. В наборе данных MNIST целевые метки в точности соответствуют выходному индексу. Следовательно, нам даже не нужно преобразовывать этот массив в строковые метки. Он используется в следующей строке кода, вызывающей функцию UI, визуализирующую результаты классификации вместе с изображениями из тестовой выборки (рис. 4.10).

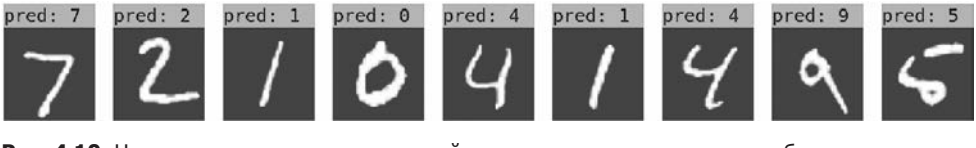


Рис. 4.10. Несколько примеров предсказаний, выполненных моделью после обучения, рядом с входными изображениями MNIST

4.3. Вне браузера: обучаем модели быстрее с помощью Node.js

В предыдущем разделе мы обучали модель в браузере и достигли безошибочности 99 % на контрольном наборе данных. В этом разделе создадим более мощную сверточную сеть, что позволит нам достичь безошибочности 99,5 %. Разумеется, за более высокую безошибочность придется заплатить потреблением моделью больших объемов памяти и вычислительных ресурсов во время как обучения, так и вывода. Увеличение вычислительных затрат более заметно во время обучения, поскольку оно включает обратное распространение ошибки, которое требует большого объема вычислений, по сравнению с прямым распространением в ходе вывода. Обучить эту бóльшую сверточную сеть в большинстве браузеров не получится: она слишком «тяжеловесна» и медленно работает.

4.3.1. Зависимости и импорты, необходимые для tfjs-node

Знакомьтесь с Node.js-версией TensorFlow.js! Она работает в среде прикладной части и не стеснена никакими ограничениями ресурсов, как вкладка браузера. В CPU-версии Node.js библиотеки TensorFlow.js (далее мы будем сокращенно называть ее *tfjs-node*) напрямую используются многопоточные математические операции, написанные на C++, те же, что и в основной Python-версии TensorFlow.js. Если же на вашей машине установлен GPU с поддержкой CUDA, tfjs-node может достичь еще большего быстродействия за счет использования математических ядер с GPU-ускорением, написанных на CUDA.

Код для нашей усовершенствованной сверточной сети MNIST находится в каталоге `mnist-node` репозитория tfjs-examples. Как и в предыдущих примерах, для доступа к коду можно использовать следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/mnist-node
```

Отличие от предыдущих примеров в том, что пример mnist-node работает в терминале, а не браузере. Для скачивания зависимостей используйте команду `yarn`.

Взглянув на содержимое файла `package.json`, вы увидите зависимость `@tensorflow/tfjs-node`. Если `@tensorflow/tfjs-node` объявляется как зависимость, `yarn` автоматически скачивает библиотеку общего пользования C++ (libtensorflow.so, libtensorflw.dylib или libtensorflow.dll в операционных системах Linux, Mac и Windows соответственно) в каталог `node_modules` для последующего использования TensorFlow.js.

По завершении работы команды `yarn` запустить обучение модели можно с помощью команды:

```
node main.js
```

Здесь мы предполагаем, что бинарные файлы node доступны по пути к исполняемым файлам, раз вы уже установили yarn (см. дополнительную информацию в приложении А).

С помощью только что описанного технологического процесса вы можете обучить нашу расширенную сверточную сеть на своем CPU. Если на вашей рабочей станции или ноутбуке есть GPU с поддержкой CUDA, вы можете также обучить модель на своем GPU. Для этого необходимо сделать следующее.

1. Установить нужную версию драйвера CUDA для своего GPU.
2. Установить набор инструментов NVIDIA CUDA — библиотеку, с помощью которой можно выполнять универсальные распараллеленные вычисления на GPU производства NVIDIA.
3. Установить CuDNN — основанную на CUDA библиотеку высокоскоростных алгоритмов глубокого обучения компании NVIDIA (подробнее шаги 1–3 описаны в приложении А).
4. В файле `package.json` заменить зависимость `@tensorflow/tfjs-node` на `@tensorflow/tfjs-node-gpu`, не меняя номера версии, поскольку выпуски этих двух пакетов синхронизированы между собой.
5. Снова выполнить команду `yarn` для скачивания библиотеки общего пользования, включающей математические операции CUDA, применяемые TensorFlow.js.
6. Заменить строку:

   ```
   require('@tensorflow/tfjs-node');
   ```

   в файле `main.js` на:

   ```
   require('@tensorflow/tfjs-node-gpu');
   ```

7. Снова запустить обучение с помощью команды:

   ```
   node main.js
   ```

Если все шаги выполнены должным образом, модель начнет обучаться с неслыханной скоростью, обычно раз в пять превышающей скорость обучения на CPU-версии (tfjs-node). Обучение модели с помощью как CPU-, так и GPU-версий tfjs-node происходит значительно быстрее, чем в браузере.

Обучение усовершенствованной сверточной сети для MNIST в tfjs-node

По завершении 20 эпох обучения модель демонстрирует итоговую безошибочность на контрольном (или проверочном) наборе данных около 99,6 % — намного больше предыдущего результата 99 %, достигнутого в разделе 4.2. Какие же различия между моделью, основанной на node, и браузерной моделью приводят к подобному скачку безошибочности? В конце концов, если обучить одну и ту же модель в tfjs-node и браузерной версии TensorFlow.js на обучающих данных, результаты не должны различаться (за исключением эффектов инициализации весов случайными значени-

ями). Для ответа на этот вопрос взглянем на описание модели, основанной на node (листинг 4.5). Она формируется в импортируемом `main.js` файле `model.js`.

Листинг 4.5. Описание усовершенствованной сверточной сети для задачи MNIST в Node.js

```
const model = tf.sequential();
model.add(tf.layers.conv2d({
  inputShape: [28, 28, 1],
  filters: 32,
  kernelSize: 3,
  activation: 'relu',
}));
model.add(tf.layers.conv2d({
  filters: 32,
  kernelSize: 3,
  activation: 'relu',
}));
model.add(tf.layers.maxPooling2d({poolSize: [2, 2]}));
model.add(tf.layers.conv2d({
  filters: 64,
  kernelSize: 3,
  activation: 'relu',
}));
model.add(tf.layers.conv2d({
  filters: 64,
  kernelSize: 3,
  activation: 'relu',
}));
model.add(tf.layers.maxPooling2d({poolSize: [2, 2]}));
model.add(tf.layers.flatten());
model.add(tf.layers.dropout({rate: 0.25}));
model.add(tf.layers.dense({units: 512, activation: 'relu'}));
model.add(tf.layers.dropout({rate: 0.5}));
model.add(tf.layers.dense({units: 10, activation: 'softmax'}));

model.summary();
model.compile({
  optimizer: 'rmsprop',
  loss: 'categoricalCrossentropy',
  metrics: ['accuracy'],
});
```

Добавляем слои дропаута для снижения переобучения

Сводка топологии модели:

```
_________________________________________________________________
Layer (type)                 Output shape              Param #
=================================================================
conv2d_Conv2D1 (Conv2D)      [null,26,26,32]           320
_________________________________________________________________
conv2d_Conv2D2 (Conv2D)      [null,24,24,32]           9248
_________________________________________________________________
max_pooling2d_MaxPooling2D1  [null,12,12,32]           0
_________________________________________________________________
conv2d_Conv2D3 (Conv2D)      [null,10,10,64]           18496
```

```
_________________________________________________________________
conv2d_Conv2D4 (Conv2D)      [null,8,8,64]             36928
_________________________________________________________________
max_pooling2d_MaxPooling2D2  [null,4,4,64]             0
_________________________________________________________________
flatten_Flatten1 (Flatten)   [null,1024]               0
_________________________________________________________________
dropout_Dropout1 (Dropout)   [null,1024]               0
_________________________________________________________________
dense_Dense1 (Dense)         [null,512]                524800
_________________________________________________________________
dropout_Dropout2 (Dropout)   [null,512]                0
_________________________________________________________________
dense_Dense2 (Dense)         [null,10]                 5130
=================================================================
Total params: 594922
Trainable params: 594922
Non-trainable params: 0
_________________________________________________________________
```

Вот список основных различий между моделью tfjs-node и браузерной моделью.

- Основанная на node модель содержит четыре слоя conv2d — на один больше, чем браузерная.
- Количество нейронов скрытого плотного слоя модели, основанной на node, больше (512), чем у его аналога в браузерной модели (100).
- В целом количество весовых коэффициентов модели, основанной на node, в 18 раз больше, чем у браузерной.
- Модель, основанная на node, включает два слоя *дропаута*, вставленных между слоем схлопывания и плотным слоем.

Благодаря первым трем различиям разрешающие возможности модели, основанной на node, больше, чем у браузерной. Но они же определяют слишком большие (для обучения в браузере с приемлемой скоростью) требования этой модели к оперативной памяти и вычислительным ресурсам. Как вы узнаете в главе 5, большие разрешающие возможности модели означают и больший риск переобучения. Этот риск в некоторой степени гасится за счет четвертого различия, а именно включения в модель слоев дропаута.

Снижение риска переобучения с помощью слоев дропаута

Дропаут — еще один новый тип слоя TensorFlow.js, с которым вам предстоит познакомиться в этой главе. Он предоставляет один из самых эффективных и широко используемых способов сокращения переобучения в глубоких нейронных сетях. Функциональность такого слоя довольно проста.

- На этапе обучения (во время вызовов метода `Model.fit()`) он случайным образом обнуляет часть элементов входного тензора (фактически отбрасывает их) и возвращает полученный результат в качестве выходного тензора. В данном примере

у слоя дропаута лишь один настроечный параметр: частота дропаута (например, два поля `rate`, как показано в листинге 4.5). Для примера допустим, что частота дропаута в этом слое задана равной `0.25`, а входной тензор представляет собой одномерный тензор со значением `[0.7, -0.3, 0.8, -0.4]`; при этом выходной тензор может оказаться, например, таким: `[0.7, -0.3, 0, -0.4]` — случайным образом было выбрано и обнулено 25 % элементов входного тензора. Аналогичным образом осуществляется подобное случайное обнуление и тензора градиентов слоя дропаута.

- На этапе вывода (во время вызовов методов `Model.predict()` и `Model.evaluate()`) слой дропаута *не* обнуляет случайным образом элементы входного тензора — входные данные просто проходят по нему без изменений (тождественное отображение).

На рис. 4.11 приведен пример работы слоя дропаута с двумерным входным тензором во время обучения и контроля.

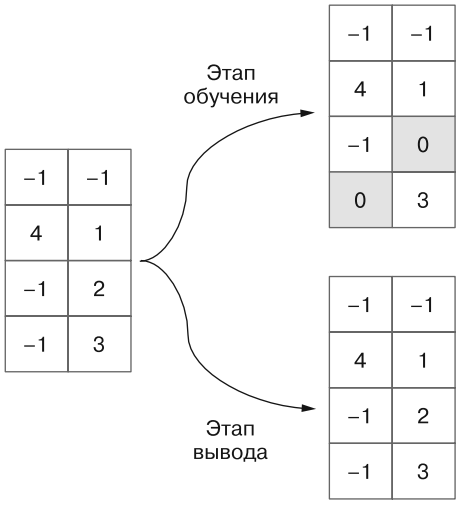


Рис. 4.11. Пример работы слоя дропаута. Здесь входной тензор двумерный, формы [4, 2]. Частота дропаута — 0,25, в результате чего 25 % (то есть два из восьми) элементов входного тензора выбираются случайным образом и обнуляются на этапе обучения. На этапе вывода слой дропаута просто пересылает данные

Может показаться странным, что такой простой алгоритм оказывается одним из лучших способов борьбы с переобучением. Почему же он столь хорош? Джефф Хинтон, который создал (помимо множества прочих вещей в сфере нейронных сетей) алгоритм дропаута, говорит, что его вдохновил используемый во многих банках механизм предотвращения мошенничества со стороны сотрудников. Он рассказывает: *«Я отправился в банк. Кассиры все время менялись, и я спросил одного из них, почему так происходит. Он ответил, что не знает, но их действительно часто переводили с места на место. Как я понял, дело в том, что обмануть банк сотрудники могут только сообща, и банк хочет лишить их этой возможности. В этот момент я осознал, что исключение случайным образом различных подмножеств нейронов для каждого примера данных предотвратит их сговор, а значит, уменьшит вероятность переобучения»*.

Если перевести все это на язык глубокого обучения, то можно выразиться так: введение шума в выходные значения слоя разрушает случайно возникшие паттерны (то, что Хинтон назвал сговором), несущественные относительно истинных паттернов данных. В упражнении 3 в конце главы вы должны будете убрать из модели, основанной на node, в файле `model.js` два слоя дропаута, обучить ее снова и проанализировать полученные значения безошибочности на обучающем, проверочном и контрольном наборах данных.

В листинге 4.6 показан основной код для обучения и оценки нашей усовершенствованной сверточной сети. Если сравнить его с кодом из листинга 4.2, сходство двух фрагментов кода становится очевидным. В основе обоих лежат вызовы `Model.fit()` и `Model.evaluate()`. Синтаксис и стиль кода одинаковы, за исключением нюансов визуализации и отображения значений потерь, показателя безошибочности и хода обучения в различных пользовательских интерфейсах (терминал и браузер).

Это демонстрирует важное свойство TensorFlow.js — фреймворка глубокого обучения на JavaScript, работающего как в клиентской, так и в прикладной части: *«В части создания и обучения моделей код, создаваемый с помощью TensorFlow.js, одинаков, независимо от того, работаете вы с браузером или Node.js»*.

Листинг 4.6. Обучение и оценка работы усовершенствованной модели в tfjs-node

```
await model.fit(trainImages, trainLabels, {
  epochs,
  batchSize,
  validationSplit
});

const {images: testImages, labels: testLabels} = data.getTestData();
const evalOutput = model.evaluate
    testImages, testLabels);
console.log('\nEvaluation result:');
console.log(
    `  Loss = ${evalOutput[0].dataSync()[0].toFixed(3)}; `+
    `Accuracy = ${evalOutput[1].dataSync()[0].toFixed(3)}`);
```

Оценка работы модели на данных, которые она еще не видела

4.3.2. Сохранение модели из Node.js и загрузка ее в браузере

Обучение модели требует расхода ресурсов CPU и GPU и занимает определенное время. Не хотелось бы, чтобы полученные результаты обучения оказались бесполезными. Если не сохранить модель, придется начинать все с начала при следующем запуске `main.js`. В этом подразделе мы покажем, как сохранить обученную модель и экспортировать сохраненную модель в файл на диске (создать так называемую *контрольную точку* (checkpoint), она же *артефакт* (artifact)). Мы также продемонстрируем, как импортировать контрольную точку в браузер, преобразовать ее обратно в модель и воспользоваться ею для вывода. Концовка функции `main()` из файла `main.js` состоит из кода сохранения модели, приведенного в листинге 4.7.

Листинг 4.7. Сохранение обученной модели в файловую систему в tfjs-node

```
if (modelSavePath != null) {
  await model.save(`file://${modelSavePath}`);
  console.log(`Saved model to path: ${modelSavePath}`);
}
```

Метод `save()` объекта `model` служит для сохранения модели в каталог файловой системы. Он принимает один аргумент — строку URL, начинающуюся с file://. Обратите внимание, что сохранение модели в файловой системе возможно благодаря использованию tfjs-node. В браузерной версии TensorFlow.js также существует API `model.save()`, но обращаться напрямую к нативной файловой системе машины он не может из-за ограничений безопасности браузера. Поэтому при использовании TensorFlow.js в браузере приходится вместо файловой системы выбирать другие места сохранения (локальное хранилище браузера и IndexedDB). Для них предусмотрены другие схемы формирования URL.

Функция `model.save()` — асинхронная, поскольку в общем случае подразумевает работу с файловым или сетевым вводом/выводом. По этой причине для вызова `save()` мы используем ключевое слово `await`. Допустим, переменной `modelSavePath` присвоено значение `/tmp/tfjs-node-mnist`. Тогда, если после завершения работы `model.save()` вывести содержимое каталога:

```
ls -lh /tmp/tfjs-node-mnist
```

список файлов окажется примерно следующим:

```
-rw-r--r-- 1 user group 4.6K Aug 14 10:38 model.json
-rw-r--r-- 1 user group 2.3M Aug 14 10:38 weights.bin
```

Здесь мы видим два файла.

- `model.json` — файл в формате JSON, содержащий сохраненную топологию модели. Под топологией мы понимаем типы слоев, составляющих модель, соответствующие параметры конфигурации (например, `filters` для слоя conv2d и `rate` для слоя дропаута), а также описание способов соединения слоев друг с другом. В случае сверточной сети MNIST эти соединения просты, поскольку модель последовательная. Далее нам предстоит встретить модели с менее тривиальными паттернами соединений, которые также можно сохранить на диск с помощью вызова `model.save()`.
- Помимо топологии модели, `model.json` содержит описание весов модели. В этой его части перечислены названия, формы и типы данных всех весовых коэффициентов модели, помимо мест, где значения этих весов хранятся. Это приводит нас ко второму файлу: `weights.bin`.
- Как ясно из его названия, `weights.bin` — это двоичный файл, в котором хранятся все значения весов модели. Он представляет собой «плоский» двоичный поток данных, без разделителей, указывающих, где начинаются и заканчиваются отдельные веса. Эта «метаинформация» доступна в той части объекта JSON из файла `model.json`, что относится к описанию весов.

Для загрузки модели с помощью tfjs-node можно воспользоваться методом `tf.loadLayersModel()`, указав ему местоположение файла `model.json` (не приводится в коде примера):

```
const loadedModel = await tf.loadLayersModel('file:///tmp/tfjs-node-mnist');
```

Метод `tf.loadLayersModel()` восстанавливает модель, десериализуя сохраненную топологию модели из файла `model.json`. Далее `tf.loadLayersModel()` считывает двоичные значения весов из файла `weights.bin` с помощью их описания из файла `model.json` и устанавливает соответствующие значения весов модели. Как и `model.save()`, метод `tf.loadLayersModel()` — асинхронный, так что при его вызове используется ключевое слово `await`. После возврата из его вызова объект `loadedModel` оказывается во всех смыслах эквивалентен модели, созданной и обученной с помощью JavaScript-кода из листингов 4.5 и 4.6. При желании можно вывести сводку топологии этой модели, вызвав метод `summary()` объекта `loadedModel`. Далее выполнить на основе этой топологии вывод с помощью метода `predict()`, оценить безошибочность модели с помощью метода `evaluate()` или даже обучить ее заново с использованием метода `fit()`. При желании можно также снова сохранить модель. Технологический процесс повторного обучения и сохранения модели окажется весьма актуальным при обсуждении переноса обучения в главе 5.

Сказанное в предыдущих абзацах применимо и к браузерной среде. Сохраненные файлы можно использовать для восстановления модели на веб-странице. Восстановленная модель поддерживает полный технологический процесс `tf.LayersModel()` с тем нюансом, что ее полное переобучение будет очень медленным и неэффективным из-за большого размера усовершенствованной сверточной сети. Единственное принципиальное различие между загрузками сохраненной модели в tfjs-node и браузере заключается в необходимости использовать отличную от file:// схему формирования URL в браузере. Обычно файлы `model.json` и `weights.bin` размещаются на HTTP-сервере в виде файлов статических ресурсов. Пусть имя вашего хоста — localhost, а файлы располагаются на сервере по пути `my/models/`. Тогда загрузить модель в браузере можно, написав такую строку кода:

```
const loadedModel = await tf.loadLayersModel('http:///localhost/my/models/
model.json');
```

При загрузке HTTP-варианта модели в браузере метод `tf.loadLayersModel()` неявно вызывает встроенную функцию `fetch` браузера. Поэтому он обладает следующими свойствами и возможностями.

- Поддерживается как http://-, так и https://-протокол.
- Поддерживаются относительные пути на сервере. На самом деле при использовании относительных путей можно опускать в URL части http:// и https://. Например, если веб-страница располагается на сервере по пути `my/index.html`, а JSON-файл модели — по пути `my/models/model.json`, можно указать относительный путь `model/model.json`:

  ```
  const loadedModel = await tf.loadLayersModel('models/model.json');
  ```

- Для указания дополнительных опций HTTP/HTTPS-запросов необходимо вместо строкового аргумента использовать метод `tf.io.browserHTTPRequest()`. Например, чтобы загрузить вместе с моделью учетные данные и заголовки, можно написать следующее:

  ```
  const loadedModel = await tf.loadLayersModel(tf.io.browserHTTPRequest(
    'http://foo.bar/path/to/model.json',
    {credentials: 'include', headers: {'key_1': 'value_1'}}));
  ```

4.4. Распознавание устной речи

До сих пор мы использовали сверточные сети для задач машинного зрения. Но органы чувств человека не ограничиваются зрением. Звуковая информация представляет собой не менее важный тип входных данных, и с ней можно работать через API браузера. Как же распознать содержимое и смысл речи и прочих звуков? Что интересно, сверточные сети подходят не только для машинного зрения, но и для связанного с аудиоданными машинного обучения.

В этом разделе мы продемонстрируем решение относительно простой задачи, связанной с обработкой звуковых данных, с помощью сверточной сети, подобной созданной нами для MNIST. Задача состоит в классификации коротких речевых сообщений примерно по 20 категориям слов. Эта задача проще, чем задачи распознавания речи, решаемые в таких устройствах, как Amazon Echo или Google Home. В частности, упомянутые системы распознавания речи требуют намного большего словарного запаса, чем в этом примере. Кроме того, они способны обрабатывать связную речь, состоящую из длинных последовательностей слов, в то время как наш пример обрабатывает произносимые по отдельности слова. Следовательно, наш пример нельзя считать «системой распознавания речи», правильнее будет описать его как «средство распознавания слов» или «средство распознавания речевых команд». Тем не менее он вполне применим на практике (например, для пользовательских интерфейсов типа hands-free и специальных возможностей). Кроме того, реализованные в примере методики глубокого обучения формируют фундамент для более продвинутых систем распознавания речи[1].

4.4.1. Спектрограммы: представление звуков в виде изображений

Как и в любом приложении глубокого обучения, чтобы разобраться в работе модели, необходимо сначала разобраться в данных. Чтобы понять, как работают сверточные сети, обрабатывающие звуковые данные, нужно разобраться с представлением звука в виде тензоров. Как вы помните из школьного курса физики, звуки речи представляют собой паттерны колебаний давления воздуха. Микрофон улавливает эти колебания и преобразует их в электрические сигналы, затем оцифровываемые с помощью звуковой карты компьютера. Современные браузеры поддерживают API *WebAudio*, способный взаимодействовать со звуковой картой и обеспечивать доступ к оцифрованным аудиосигналам в режиме реального времени (с разрешения пользователя). Таким образом, с точки зрения программиста на JavaScript, звуки — это массивы вещественных чисел. В глубоком обучении подобные массивы чисел обычно представляются в виде одномерных тензоров.

Вероятно, вы недоумеваете, как сверточные сети, подобные приведенным выше, могут работать с одномерными тензорами. Разве они не предназначены для работы с тензорами по крайней мере ранга 2? Ключевые слои сверточной сети, включая

[1] *Collobert R., Puhrsch C., Synnaeve G.* Wav2Letter: An End-to-End ConvNet-based Speech Recognition System // submitted 13 Sept. 2016. https://arxiv.org/abs/1609.03193.

conv2d и maxPooling2d, используют в своей работе пространственные отношения в двумерных пространствах. Оказывается, звуки *можно* представить в виде особых изображений, так называемых *спектрограмм* (spectrogram). Спектрограммы не только позволяют обрабатывать звуки с помощью сверточных нейронных сетей, но и могут применяться вне сферы глубокого обучения.

Как демонстрирует рис. 4.12, спектрограмма — это двумерный массив чисел, который можно представить в виде изображения в оттенках серого, примерно так, как и изображения MNIST. Горизонтальное измерение — время, а вертикальное — частота. Вертикальные полосы спектрограммы отражают *спектр* (spectrum) частот в пределах короткого временного окна. Спектр — это разбиение звукового сигнала на различные частотные компоненты, которые можно рассматривать как отдельные «тона». Подобно тому как свет можно разбить с помощью призмы на множество цветов, звук можно разбить на отдельные частоты с помощью математической операции, именуемой *преобразованием Фурье* (Fourier transform). По сути, спектрограмма описывает изменения частотного содержимого звукового сигнала за несколько последовательных, узких временных окон (обычно длиной порядка 20 миллисекунд).

На рис. 4.12 каждая из полос по временной оси (столбец изображения) представляет собой краткий промежуток (кадр) времени; а каждый срез по оси частоты (строка изображения) соответствует конкретному узкому диапазону частот (тону). Значения пикселов изображения соответствуют относительной мощности звукового сигнала в данном диапазоне частот в конкретный момент времени. Спектрограммы здесь изображены так, что более темному оттенку серого соответствует бóльшая мощность. Различные звуки речи обладают разными отличительными признаками. Например, свистящие звуки, такие как z и s, характеризуются квазистационарным состоянием мощности, сосредоточенной на частотах выше 2–3 кГц; гласные звуки, например e и o, характеризуются горизонтальными полосами (всплесками мощности) на нижнем конце спектра частот (< 3 кГц). Эти всплески мощности в акустике называются *формантами*. У различных гласных разные частоты формант. Глубокая сверточная сеть может использовать все эти отличительные признаки различных звуков речи для распознавания слов.

Спектрограммы удобны в качестве представления звуковых сигналов по следующим причинам. Во-первых, они экономят место: количество чисел с плавающей точкой в спектрограмме обычно в несколько раз меньше, чем в исходной волновой форме. Во-вторых, в некотором смысле спектрограммы отражают работу слуха в биологии. Анатомическая структура во внутреннем ухе — улитка — по существу, выполняет биологическое «преобразование Фурье». Она раскладывает звуки на отдельные частоты, улавливаемые различными наборами слуховых нейронов. В-третьих, представление звуков речи в виде спектрограммы позволяет проще отличать типы звуков речи друг от друга. Это хорошо видно из примеров спектрограмм на рис. 4.12: у гласных и согласных совершенно разные отличительные признаки на спектрограммах. Десятки лет назад, еще до того, как машинное обучение стало широко применяться, специалисты по распознаванию речи пытались вручную сформировать правила, которые бы позволяли обнаруживать на спектрограммах различные гласные и согласные. Глубокое обучение дает возможность сэкономить силы и нервы, необходимые для подобной «ручной работы».

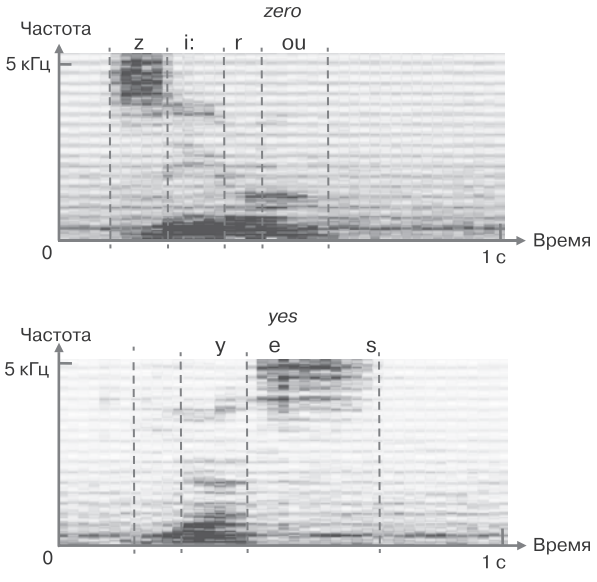


Рис. 4.12. Примеры спектрограмм произнесенных по отдельности слов zero и yes. Спектрограмма представляет собой совместное частотно-временное представление звукового сигнала. Ее можно рассматривать как представление звука в виде изображения

Давайте теперь на минуту остановимся и задумаемся. Глядя на изображения MNIST на рис. 4.1 и спектрограммы речи на рис. 4.12, вы не могли не обратить внимания на сходство этих двух наборов данных. Оба содержат паттерны в двумерном пространстве признаков, заметные тренированному глазу. Точное местоположение, размер и прочие нюансы признаков в обоих наборах данных отличаются некоторой степенью случайности. Количество возможных классов для MNIST равно 10, а для набора данных речевых команд — 20 (десять цифр от 0 до 9, *up*, *down*, *left*, *right*, *go*, *stop*, *yes* и *no*, а также категории «неизвестных» слов и фонового шума). Именно благодаря этим принципиальным сходствам между наборами данных сверточные сети подходят для задачи распознавания речи.

Но есть между этими двумя наборами данных и существенные различия. Во-первых, аудиозаписи в наборе данных речевых команд несколько зашумлены, как вы можете видеть по пятнышкам темных пикселов, не имеющих отношения к звукам речи в примере на рис. 4.12. Во-вторых, все спектрограммы в наборе данных речевых команд имеют размер 43×232, что значительно превышает размеры отдельных изображений MNIST, составлявшие 28×28. Размер спектрограммы асимметричен по измерениям времени и частоты. Эти отличия найдут свое отражение в сверточной сети, которую мы будем использовать для нашего звукового набора данных.

Код для описания и обучения сверточной сети распознавания голосовых команд находится в репозитории tfjs-models. Для получения этого кода можно использовать следующие команды:

```
git clone https://github.com/tensorflow/tfjs-models.git
cd speech-commands/training/browser-fft
```

Код создания и компиляции модели мы инкапсулировали в функции `createModel()` из файла `model.ts` (листинг 4.8).

Листинг 4.8. Сверточная сеть для классификации спектрограмм речевых команд

```
function createModel(inputShape: tf.Shape, numClasses: number) {
  const model = tf.sequential();
  model.add(tf.layers.conv2d({
    filters: 8,
    kernelSize: [2, 8],
    activation: 'relu',
    inputShape
  }));
  model.add(tf.layers.maxPooling2d({poolSize: [2, 2], strides: [2, 2]}));
  model.add(      tf.layers.conv2d({
        filters: 32,
        kernelSize: [2, 4],
        activation: 'relu'
      }));
  model.add(tf.layers.maxPooling2d({poolSize: [2, 2], strides: [2, 2]}));
  model.add(
      tf.layers.conv2d({
        filters: 32,
        kernelSize: [2, 4],
        activation: 'relu'
      }));
  model.add(tf.layers.maxPooling2d({poolSize: [2, 2], strides: [2, 2]}));
  model.add(
      tf.layers.conv2d({
        filters: 32,
        kernelSize: [2, 4],
        activation: 'relu'
      }));
  model.add(tf.layers.maxPooling2d(
          {poolSize: [2, 2], strides: [1, 2]}));
  model.add(tf.layers.flatten());
  model.add(tf.layers.dropout({rate: 0.25}));
  model.add(tf.layers.dense({units: 2000, activation: 'relu'}));
  model.add(tf.layers.dropout({rate: 0.5}));
  model.add(tf.layers.dense({units: numClasses, activation: 'softmax'}));

  model.compile({
    loss: 'categoricalCrossentropy',
    optimizer: tf.train.sgd(0.01),
    metrics: ['accuracy']
  });
  model.summary();
  return model;
}
```

Повторяющиеся «лейтмотивы» свертки и субдискретизации с выбором максимального значения

Начало многослойного перцептрона

Для снижения вероятности переобучения используется слой дропаута

Задаем функцию потерь и метрику для многоклассовой классификации

Топология нашей сверточной сети для аудиоданных во многом схожа со сверточной сетью для MNIST. Последовательная модель начинается с нескольких

повторяющихся «мотивов» слоев conv2d в сочетании со слоями maxPooling2d. Часть модели, связанная со сверткой-субдискретизацией, заканчивается на слое схлопывания, за которым располагается многослойный перцептрон. MLP включает два плотных слоя. Функция активации скрытого плотного слоя — ReLU, а последнего (выходного) слоя — многомерная логистическая функция активации, подходящая для задачи классификации. При компиляции модели в качестве функции потерь указана `categoricalCrossentropy`, а во время обучения и оценки генерируется метрика безошибочности. Все точно так же, как и в сверточных сетях для MNIST, поскольку оба набора данных требуют многоклассовой классификации. Сверточная сеть для обработки аудиоданных также отличается от сети MNIST интересными особенностями. В частности, форма свойств `kernelSize` слоев conv2d — прямоугольная (например, `[2, 8]`), а не квадратная, чтобы соответствовать неквадратной форме спектрограмм, измерение частоты которых больше, чем временно́е.

Для обучения модели необходимо сначала скачать набор данных речевых команд. Он был создан на основе набора данных речевых команд, собранного Питом Уорденом, инженером из команды Google Brain (см. https://www.tensorflow.org/tutorials/audio/simple_audio), и преобразован в подходящий для браузера формат спектрограмм:

```
curl -fSsL https://storage.googleapis.com/learnjs-data/speech-commands/
  speech-commands-data- v0.02-browser.tar.gz
  -o speech-commands-data-v0.02-browser.tar.gz &&
tar xzvf speech-commands-data-v0.02-browser.tar.gz
```

Эти команды скачивают и извлекают браузерную версию набора данных речевых команд. После извлечения данных можно запустить процесс обучения с помощью следующей команды:

```
yarn
yarn train \
    speech-commands-data-browser/ \
    /tmp/speech-commands-model/
```

Первый аргумент команды `yarn train` указывает на место размещения обучающих данных. Следующие аргументы задают путь сохранения JSON-файла модели, а также файла весов и JSON-файла метаданных. Как и в случае усовершенствованной сверточной сети MNIST, мы обучаем сверточную сеть для аудиоданных в tfjs-node, что позволяет использовать GPU. А поскольку размеры набора данных и модели намного больше, чем у сверточной сети MNIST, обучение займет больше времени (порядка нескольких часов). Процесс можно значительно ускорить при наличии поддерживающего CUDA GPU, слегка изменив команду так, чтобы использовать tfjs-node-gpu вместо tfjs-node по умолчанию (выполняемого только на CPU). Для этого просто добавьте в предыдущую команду флаг `--gpu`:

```
yarn train \
--gpu \
    speech-commands-data-browser/ \
    /tmp/speech-commands-model/
```

По завершении обучения модель должна достичь итоговой безошибочности на проверочном (контрольном) наборе данных примерно 94 %.

Обученная модель сохраняется по пути, заданному с помощью флага в предыдущей команде. Как и в случае сверточной сети для MNIST, обученной нами с помощью tfjs-node, сохраненную модель позже можно загрузить в браузере. Впрочем, для получения данных с микрофона и предварительного их преобразования в подходящий для использования моделью формат необходимы определенные знания API WebAudio. Для вашего удобства мы написали класс-адаптер, который умеет не только загружать обученную сверточную сеть для аудиоданных, но и выполнять ввод и предварительную обработку данных. Если вам интересны механизмы конвейера ввода аудиоданных, можете заглянуть в исходный код из репозитория Git tfjs-model, расположенный в каталоге `speech-commands/src`. Этот адаптер доступен через систему управления пакетами npm под названием `@tensorflow-models/speech-commands`. В листинге 4.9 показан весьма минималистичный пример использования этого класса для онлайн-распознавания слов речевых команд в браузере.

В каталоге `speech-commands/demo` репозитория tfjs-models вы найдете более полноценный пример применения этого пакета. Для клонирования и запуска демонстрационного примера выполните следующие команды в каталоге `speech-commands`:

```
git clone https://github.com/tensorflow/tfjs-models.git
cd tfjs-models/speech-commands
yarn && yarn publish-local
cd demo
yarn && yarn link-local && yarn watch
```

Команда `yarn watch` автоматически открывает новую вкладку в браузере, используемом по умолчанию. Чтобы увидеть средство распознавания речевых команд в действии, убедитесь, что на вашей машине доступен микрофон (он встроен в большинство ноутбуков). Каждое распознанное слово из словаря будет сразу отображаться на экране вместе с содержащей его спектрограммой длительностью одна секунда. Перед вами браузерное средство распознавания отдельных слов в действии, основанное на API WebAudio и глубокой сверточной сети. Но ведь оно не способно распознавать связную речь, наделенную грамматикой, правда? Для подобного нам понадобятся базовые элементы других типов нейронных сетей, способные обрабатывать последовательную информацию. Мы вернемся к этому вопросу в главе 8.

Листинг 4.9. Пример использования модуля @tensorflow-models/speech-commands

Импорт модуля speech-commands. Не забудьте указать его в качестве зависимости в файле package.json

```
import * as SpeechCommands from
    '@tensorflow-models/speech-commands';

const recognizer =
    SpeechCommands.create('BROWSER_FFT');

console.log(recognizer.wordLabels());
```

Создает экземпляр средства распознавания речевых команд, использующего встроенное в браузер быстрое преобразование Фурье (FFT)

Можете проверить, какие метки слов (включая метки для фонового шума и «неизвестных» слов) модель способна распознавать

```
recognizer.listen(result => {
  let maxIndex;
  let maxScore = -Infinity;
  result.scores.forEach((score, i) => {
    if (score > maxScore) {
      maxIndex = i;
      maxScore = score;
    }
  });
  console.log(`Detected word ${recognizer.wordLabels()[maxIndex]}`);
}, {
  probabilityThreshold: 0.75
});

setTimeout(() => recognizer.stopStreaming(), 10e3);
```

result.scores содержит оценки вероятности, соответствующие recognizer.wordLabels()

Ищем индекс слова с наивысшей оценкой вероятности

Останавливаем потоковое онлайн-распознавание через 10 секунд

Запускаем потоковое онлайн-распознавание. Первый аргумент — обратный вызов, вызываемый при каждом распознавании известного слова, не являющегося фоновым шумом, с вероятностью выше порогового значения (в данном случае 0,75)

Упражнения

1. Сверточная сеть для классификации изображений MNIST в браузере (см. листинг 4.1) включает две группы слоев conv2d и maxPooling2d. Модифицируйте код так, чтобы осталась только одна группа. Ответьте на следующие вопросы.
   А. Как это изменение повлияло на общее число обучаемых параметров сети?
   Б. Как изменение повлияло на скорость обучения?
   В. Как изменение повлияло на итоговую безошибочность сети после обучения?
2. Это упражнение аналогично упражнению 1. Но вместо изменения количества групп слоев conv2d-maxPooling2d поэкспериментируйте с количеством плотных слоев MLP-части сверточной сети в листинге 4.1. Как изменятся общее число параметров, скорость обучения и итоговая безошибочность, если удалить первый плотный слой, оставив только второй (выходной)?
3. Удалите слой дропаута из сверточной сети mnist-node (см. листинг 4.5) и посмотрите, как это повлияет на процесс обучения и итоговый показатель безошибочности на контрольном наборе данных. Почему так происходит и что это означает?
4. Для наработки практических навыков извлечения данных изображений из графических и видеоэлементов веб-страницы с помощью метода `tf.browser.fromPixels()` попробуйте сделать следующее.
   А. С помощью метода `tf.browser.fromPixels()` получите тензорное представление цветного изображения JPG из элемента `img`.
      - Каковы высота и ширина тензора изображения, возвращенного методом `tf.browser.fromPixels()`? Что определяет эти высоту и ширину?

- Измените размер изображения до 100 × 100 (высота × ширина) с помощью метода `tf.image.resizeBilinear()`.
- Повторите предыдущий шаг, но теперь воспользуйтесь другой функцией для измерения размера: `tf.image.resizeNearestNeighbor()`. Видите какие-нибудь различия между результатами работы этих двух функций изменения размера?

Б. Создайте холст HTML и нарисуйте на нем произвольные фигуры с помощью функций наподобие `rect()`. Либо, если хотите, воспользуйтесь более продвинутыми библиотеками, например d3.js или three.js, для рисования на нем более сложных двумерных и трехмерных фигур. Далее извлеките с этого холста данные изображения в виде тензора с помощью метода `tf.browser.fromPixels()`.

Резюме

- Сверточные сети выделяют из входных изображений двумерные пространственные признаки с помощью иерархии последовательных слоев conv2d и maxPooling2d.
- Слои conv2d — это многоканальные настраиваемые пространственные фильтры. Благодаря своим свойствам локальности и единства параметров они обладают широкими возможностями выделения признаков и эффективного преобразования представлений.
- Слои maxPooling2d уменьшают размер тензора входного изображения за счет вычисления максимума по окну фиксированного размера, тем самым снижая степень зависимости от конкретной позиции.
- «Стопка» слоев conv2d-maxPooling2d сверточной сети обычно завершается слоем схлопывания, за которым следует состоящий из плотных слоев MLP, предназначенный для классификации или регрессии.
- Из-за ограниченности ресурсов браузер подходит для обучения только маленьких моделей. Для обучения бóльших моделей мы рекомендуем использовать tfjs-node — версию TensorFlow.js для Node.js; tfjs-node умеет использовать те же распараллеленные ядра CPU и GPU, что и Python-версия TensorFlow.
- Чем больше разрешающие возможности модели, тем выше риск переобучения. Снизить риск переобучения можно, добавив в сверточную сеть слои дропаута. Слои дропаута во время обучения обнуляют заданную часть входных элементов, выбираемых случайным образом.
- Сверточные сети пригодны не только для решения задач машинного зрения. С их помощью можно достичь высокой степени безошибочности классификации аудиосигналов, представленных в виде спектрограмм.

5 Перенос обучения: переиспользование предобученных нейронных сетей

В этой главе

- Что такое перенос обучения и почему во многих задачах лучше использовать этот метод, а не обучать модели с нуля.
- Как использовать все возможности выделения признаков в современных предобученных сверточных слоях, преобразуя их из Keras в TensorFlow.js.
- Подробное описание механизмов переноса обучения, включая блокировку слоев, создание новых переносимых верхушек и тонкую подстройку.
- Обучение в TensorFlow.js простой модели для обнаружения объектов с помощью переноса обучения.

В главе 4 обсуждалось обучение сверточных сетей для классификации изображений. Теперь представьте себе такой сценарий работы: сеть классификации рукописных цифр демонстрирует плохие результаты для части пользователей, поскольку их почерк сильно отличается от того, каким написаны исходные обучающие данные. Можно ли повысить для них качество работы модели на основе небольшого количества (скажем, 50 примеров) полученных от них же данных? И еще один сценарий работы: интернет-магазин хочет автоматически классифицировать загружаемые пользователями изображения товаров. Но среди общедоступных сетей (таких как MobileNet[1]) нет обученных на изображениях из подобной узкой предметной области.

[1] *Howard A. G. et al.* MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications // submitted 17 Apr. 2017. https://arxiv.org/abs/1704.04861.

Можно ли воспользоваться общедоступной моделью изображений для решения такой специфической задачи классификации, если имеется умеренное количество (скажем, несколько сотен) маркированных изображений?

К счастью, решать подобные задачи можно с помощью методики *переноса обучения* (transfer learning), которой и посвящена данная глава.

5.1. Переиспользование предобученных моделей

По существу, перенос обучения представляет собой ускорение новой задачи обучения за счет переиспользования результатов предыдущего обучения. При этом используется модель, обученная на наборе данных выполнять *другую, но схожую* задачу ML. Такую уже обученную модель называют *базовой* (base model). Иногда перенос обучения означает повторное обучение базовой модели, а иногда — создание новой модели на основе базовой. Новую модель мы будем называть *перенесенной* (transfer model). Как демонстрирует рис. 5.1, необходимый для повторного обучения объем данных намного меньше, по сравнению с массивом данных, на котором обучалась базовая модель (как в двух примерах выше). Перенос обучения обычно требует меньше времени и ресурсов, чем процесс обучения базовой модели. Благодаря этому он осуществим в TensorFlow.js в среде с ограниченными ресурсами, например в браузере. Поэтому для изучающих TensorFlow.js перенос обучения — очень важная тема.

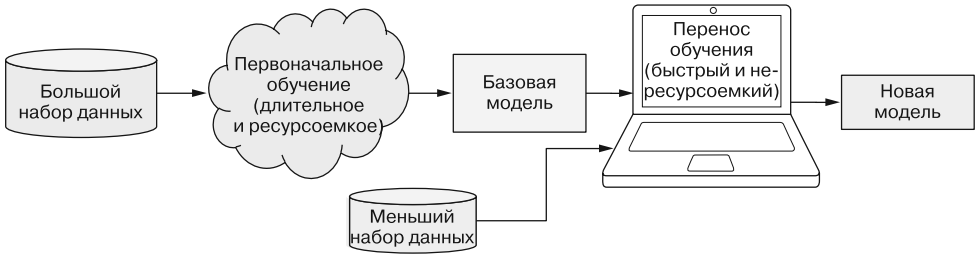


Рис. 5.1. Общая схема переноса обучения. Базовая модель обучается на большом наборе данных, причем этот начальный процесс часто требует много времени и ресурсов. Затем базовая модель обучается повторно, возможно становясь частью новой модели. Для повторного обучения требуется намного меньше данных и вычислений, чем для первоначального. Его можно выполнить и на оконечном устройстве с запущенным TensorFlow.js, например на ноутбуке или телефоне

Ключевая фраза «другую, но схожую» в определении переноса обучения при разных ситуациях означает разные вещи.

- В первом, упомянутом в начале главы сценарии уже существующая модель адаптируется к данным конкретного пользователя. И хотя они отличаются от данных исходного обучающего набора, задача остается той же — классификация изображений по десяти категориям (цифрам). Эта разновидность переноса обучения называется *адаптацией модели* (model adaptation).

- В прочих задачах переноса обучения используются целевые признаки (метки), отличающиеся от исходных. К этой разновидности относится задача классификации изображений товаров, упомянутая в начале данной главы.

В чем преимущества переноса обучения по сравнению с обучением новой модели с нуля? Их два.

- Перенос обучения эффективнее в смысле объемов как требуемых данных, так и вычислений.
- Перенос обучения далее развивает результаты предыдущего обучения, переиспользуя возможности выделения признаков базовой модели.

Перенос обучения сохраняет эти преимущества независимо от типа решаемой задачи (например, классификации или регрессии). В первом пункте при переносе обучения используются усвоенные весовые коэффициенты из базовой модели (или некое их подмножество). В результате требуется меньше обучающих данных и обучение сходится к заданному уровню безошибочности за меньшее время по сравнению с обучением новой модели с нуля. В этом перенос обучения напоминает учебу людей: когда вы научились решать какую-либо задачу (например, играть в карточную игру), научиться в будущем решать аналогичные задачи (скажем, играть в схожие карточные игры) сможете намного проще и быстрее. Количество сэкономленного времени не так велико для нейронных сетей наподобие созданной нами для MNIST сверточной сети. Однако для более масштабных моделей, обучаемых на бóльших наборах данных (например, сверточных сетей промышленного масштаба, обучаемых на терабайтах данных изображений), экономия оказывается существенной.

Что касается второго пункта, основная идея переноса обучения и заключается в переиспользовании результата предыдущего обучения. Исходная нейронная сеть, обученная на очень большом наборе данных, прекрасно решает задачу выделения полезных признаков из первоначальных входных данных. Признаки принесут пользу и при решении задачи, для которой нужен перенос обучения, если новые данные не слишком отличаются от первоначальных. Исследователи собрали множество обширных наборов данных для часто встречающихся предметных областей машинного обучения. В сфере машинного зрения есть набор данных ImageNet[1] с миллионами маркированных изображений, относящихся примерно к тысяче различных категорий. Исследователи в сфере DL обучили на наборе данных ImageNet множество глубоких сверточных сетей, включая ResNet, Inception и MobileNet (последней мы скоро займемся). Благодаря большому количеству и разнообразию изображений в наборе данных ImageNet обученные на нем сверточные сети отлично подходят для выделения признаков для распространенных изображений. Они удобны для работы с маленькими наборами данных наподобие упоминавшихся выше, на которых нельзя обучить настолько эффективные средства выделения признаков. Перенос обучения может пригодиться и в других предметных областях. Например, в сфере обработки

[1] Пусть его название не вводит вас в заблуждении, ImageNet — это набор данных, а не нейронная сеть.

естественного языка существуют вложения слов (векторные представления всех часто встречающихся слов языка), обученные на огромных корпусах текста из миллиардов слов. Эти вложения очень удобны для задач понимания языка, в которых доступны лишь намного меньшие текстовые наборы данных. А теперь наконец взглянем на пример того, как перенос обучения работает на практике.

5.1.1. Перенос обучения при совместимых формах выходных сигналов: блокировка слоев

Начнем с относительно простого примера: обучим сверточную сеть на пяти первых цифрах набора данных MNIST (от 0 до 4). Затем воспользуемся полученной моделью для распознавания оставшихся пяти цифр (от 5 до 9), которые модель не видела во время первоначального обучения. Несмотря на некоторую искусственность, этот пример хорошо иллюстрирует основной технологический процесс переноса обучения. Получить и запустить код примера можно с помощью следующих команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/mnist-transfer-cnn
yarn && yarn watch
```

На открывшейся демостранице запустите процесс обучения, нажав кнопку Retrain. Вы увидите, как обучение достигает безошибочности 96 % на новом наборе из пяти цифр (от 5 до 9), что на относительно мощном ноутбуке занимает около 30 секунд — это намного быстрее, чем вариант без переноса обучения (а именно, обучение новой модели с нуля). Взглянем, как это реализовано, шаг за шагом.

Чтобы не отвлекать ваше внимание от ключевых этапов технологического процесса, в примере мы загружаем предобученную базовую модель с HTTP-сервера, вместо того чтобы обучать ее с нуля. Как вы помните из подраздела 4.3.3, TensorFlow.js предоставляет метод `tf.loadLayersModel()` для загрузки предобученных моделей. Вызываем этот метод в файле `loader.js`:

```
const model = await tf.loadLayersModel(url);
model.summary();
```

Сводка топологии модели выглядит так, как показано на рис. 5.2. Как видите, модель состоит из 12 слоев[1]. Все ее 600 000 или около того весовых параметров — обучаемые, как и во встречавшихся нам ранее моделях TensorFlow.js. Обратите

[1] Возможно, вы еще не сталкивались со слоями активации, присутствующими в этой модели. Слои активации — это простые слои, всего лишь применяющие к входному сигналу функцию активации (например, ReLU или многомерную логистическую функцию активации). Пусть дан плотный слой с функцией активации по умолчанию (линейной); присоединение к нему слоя активации эквивалентно использованию плотного слоя с соответствующей функцией активации (не той, что по умолчанию). Именно это мы и делали в примерах из главы 4, но на практике встречаются оба варианта. В TensorFlow.js подобную топологию можно создать с помощью следующего кода: const model = tf.sequential(); model.add(tf.layers.dense({untis: 5, inputShape})); model.add(tf.layers.activation({activation: 'relu'}).

внимание, что метод `loadLayersModel()` загружает не только топологию модели, но и все значения ее весовых коэффициентов. В результате загруженная модель сразу готова к предсказанию классов цифр от 0 до 4. Впрочем, мы будем использовать ее не для этого, а для распознавания новых цифр (от 5 до 9).

```
Layer (type)                 Output shape              Param #
=================================================================
conv2d_1 (Conv2D)            [null,26,26,32]           320
_________________________________________________________________
activation_1 (Activation)    [null,26,26,32]           0
_________________________________________________________________
conv2d_2 (Conv2D)            [null,24,24,32]           9248
_________________________________________________________________
activation_2 (Activation)    [null,24,24,32]           0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 [null,12,12,32]           0
_________________________________________________________________
dropout_1 (Dropout)          [null,12,12,32]           0
_________________________________________________________________
flatten_1 (Flatten)          [null,4608]               0
_________________________________________________________________
dense_1 (Dense)              [null,128]                589952
_________________________________________________________________
activation_3 (Activation)    [null,128]                0
_________________________________________________________________
dropout_2 (Dropout)          [null,128]                0
_________________________________________________________________
dense_2 (Dense)              [null,5]                  645
_________________________________________________________________
activation_4 (Activation)    [null,5]                  0
=================================================================
Total params: 600165
Trainable params: 600165
Non-trainable params: 0
_________________________________________________________________
```

Во время переноса обучения будут заблокированы (не будут обучаться)

Рис. 5.2. Сводка топологии сверточной сети для распознавания изображений MNIST и переноса обучения

Глядя на функцию обратного вызова для кнопки Retrain (в функции `retrainModel()` файла `index.js`), вы заметите несколько строк кода, присваивающих значение `false` свойству `trainable` первых семи слоев модели, если выбрана опция Freeze Feature Layers (а она выбрана по умолчанию).

Что происходит в результате? По умолчанию свойство `trainable` всех слоев модели непосредственно после ее загрузки с помощью метода `loadLayersModel()` или обучения с нуля равно `true`. Свойство `trainable` используется во время обучения (то есть вызова метода `fit()` или `fitDataset()`) и указывает оптимизатору, нужно ли обновлять весовые коэффициенты слоя. По умолчанию весовые коэффициенты всех слоев модели обновляются во время обучения. Если же присвоить свойству `trainable` значение `false` для части слоев модели, весовые коэффициенты этих слоев *не* будут обновляться во время обучения. Говоря на языке TensorFlow.js, эти слои станут *необучаемыми* (untrainable) или *заблокированными*

(frozen). Код в листинге 5.1 блокирует первые семь слоев модели, от входного слоя conv2d до слоя схлопывания, в то время как несколько последних слоев (плотные) остаются обучаемыми.

Листинг 5.1. «Блокировка» нескольких первых слоев сверточной сети для переноса обучения

```
const trainingMode = ui.getTrainingMode();
if (trainingMode === 'freeze-feature-layers') {
  console.log('Freezing feature layers of the model.');
  for (let i = 0; i < 7; ++i) {
    this.model.layers[i].trainable = false;
  }
} else if (trainingMode === 'reinitialize-weights') {
  const returnString = false ;
  this.model = await tf.models.modelFromJSON({
    modelTopology: this.model.toJSON(null, returnString)
  });

}
this.model.compile({
  loss: 'categoricalCrossentropy',
  optimizer: tf.train.adam(0.01),
  metrics: ['acc'],
});

this.model.summary();
```

Блокирует слой

Создает новую модель с той же топологией, что и у старой, но с заново заданными значениями весовых коэффициентов

Если сначала не скомпилировать модель, во время вызовов fit() блокировка не сработает

Снова выводим сводку топологии модели после вызова compile(). В ней должно быть видно, что часть весовых коэффициентов модели стали необучаемыми

Впрочем, недостаточно изменить одно свойство `trainable` слоя: если просто поменять его значение и сразу вызвать метод `fit()` модели, веса соответствующих слоев все равно будут обновляться во время вызова `fit()`. Необходимо перед вызовом `Model.fit()` вызвать метод `Model.compile()`, чтобы изменения свойства `trainable` вступили в силу, как это сделано в листинге 5.1. Мы уже упоминали, что вызов `compile()` задает настройки оптимизатора, функцию потерь и метрики. Однако этот метод также позволяет модели актуализировать список обновляемых во время этих вызовов весовых переменных. После вызова `compile()` мы снова выводим сводку топологии модели с помощью `summary()`. Как видно при сравнении старой топологии с рис. 5.2 с новой, часть весовых коэффициентов модели стали необучаемыми:

```
Total params: 600165
Trainable params: 590597
Non-trainable params: 9568
```

Можете сами убедиться, что число необучаемых параметров, 9568, равно сумме количества весовых коэффициентов в единственных двух заблокированных слоях с весами (двух слоях conv2d). Обратите внимание, что некоторые из заблокированных нами слоев не содержат весовых коэффициентов (например, maxPooling2d и слой схлопывания), а потому не вносят никакого вклада в число необучаемых параметров при блокировании.

Сам код переноса обучения приведен в листинге 5.2. В нем используется тот же метод `fit()`, что и при обучении моделей с нуля. В этом вызове для оценки безошибочности модели на еще не виденных ею данных применяется поле `validationData`.

Кроме того, мы подключаем к вызову `fit()` две функции обратного вызова, одну для обновления индикатора хода выполнения в UI, а вторую — для построения графиков кривых потерь и безошибочности с помощью модуля tfjs-vis (больше подробностей вы найдете в главе 7). Это демонстрирует еще не упоминавшийся нами аспект API `fit()`: в вызов `fit()` можно передать функцию обратного вызова или даже массив из нескольких функций обратного вызова. Во втором случае во время обучения вызываются все функции обратного вызова (в порядке, указанном в массиве).

Листинг 5.2. Перенос обучения с помощью функции Model.fit()

```
await this.model.fit(this.gte5TrainData.x, this.gte5TrainData.y, {
  batchSize: batchSize,
  epochs: epochs,
  validationData: [this.gte5TestData.x, this.gte5TestData.y],
  callbacks: [
    ui.getProgressBarCallbackConfig(epochs),
    tfVis.show.fitCallbacks(surfaceInfo, ['val_loss', 'val_acc'], {
      zoomToFit: true,
      zoomToFitAccuracy: true,
      height: 200,
      callbacks: ['onEpochEnd'],
    }),
  ]
});
```

В функцию fit() можно передавать несколько функций обратного вызова

Строим графики кривых потерь и безошибочности во время переноса обучения с помощью модуля tfjs-vis

Насколько хороши результаты переноса обучения? Как видите в блоке А на рис. 5.3, после десяти эпох обучения, занимающих примерно 15 секунд на относительно современном ноутбуке, безошибочность достигает примерно 0,968 — совсем неплохо. Но как сравнить этот результат с результатом модели, обучаемой с нуля? Один из способов сравнения — заново задать случайным образом начальные значения весов предобученной модели непосредственно перед вызовом `fit()`. Именно это и произойдет, если перед нажатием кнопки Retrain вы выберете опцию Reinitialize Weights из раскрывающегося меню Training Mode. Результат показан в блоке Б на рис. 5.3.

Как можно видеть из сравнения блока А с блоком Б, при повторной инициализации случайным образом значений весов модели начальное значение функции потерь оказывается значительно выше (0,36 по сравнению с 0,30), а безошибочность — ниже (0,88 вместо 0,91). У заново инициализированной модели итоговая безошибочность на проверочном наборе также ниже (~0,954), чем у модели, переиспользующей весовые коэффициенты базовой модели (~0,968). Эти различия наглядно демонстрируют преимущества переноса обучения: благодаря переиспользованию весовых коэффициентов в начальных слоях (слоях, служащих для выделения признаков) модели она получает определенный гандикап по сравнению с обучением с нуля. А все потому, что встречающиеся в задаче перенесенного обучения данные аналогичны данным, на которых обучалась исходная модель. Изображения цифр с 5 до 9 во многом схожи с изображениями цифр от 0 до 4: все представляют собой изображения в оттенках серого на черном фоне со схожими визуальными паттернами (штрихами схожей толщины и кривизны). Поэтому признаки, которые модель научилась выделять из цифр от 0 до 4, оказываются пригодны и для обучения классификации новых цифр (от 5 до 9).

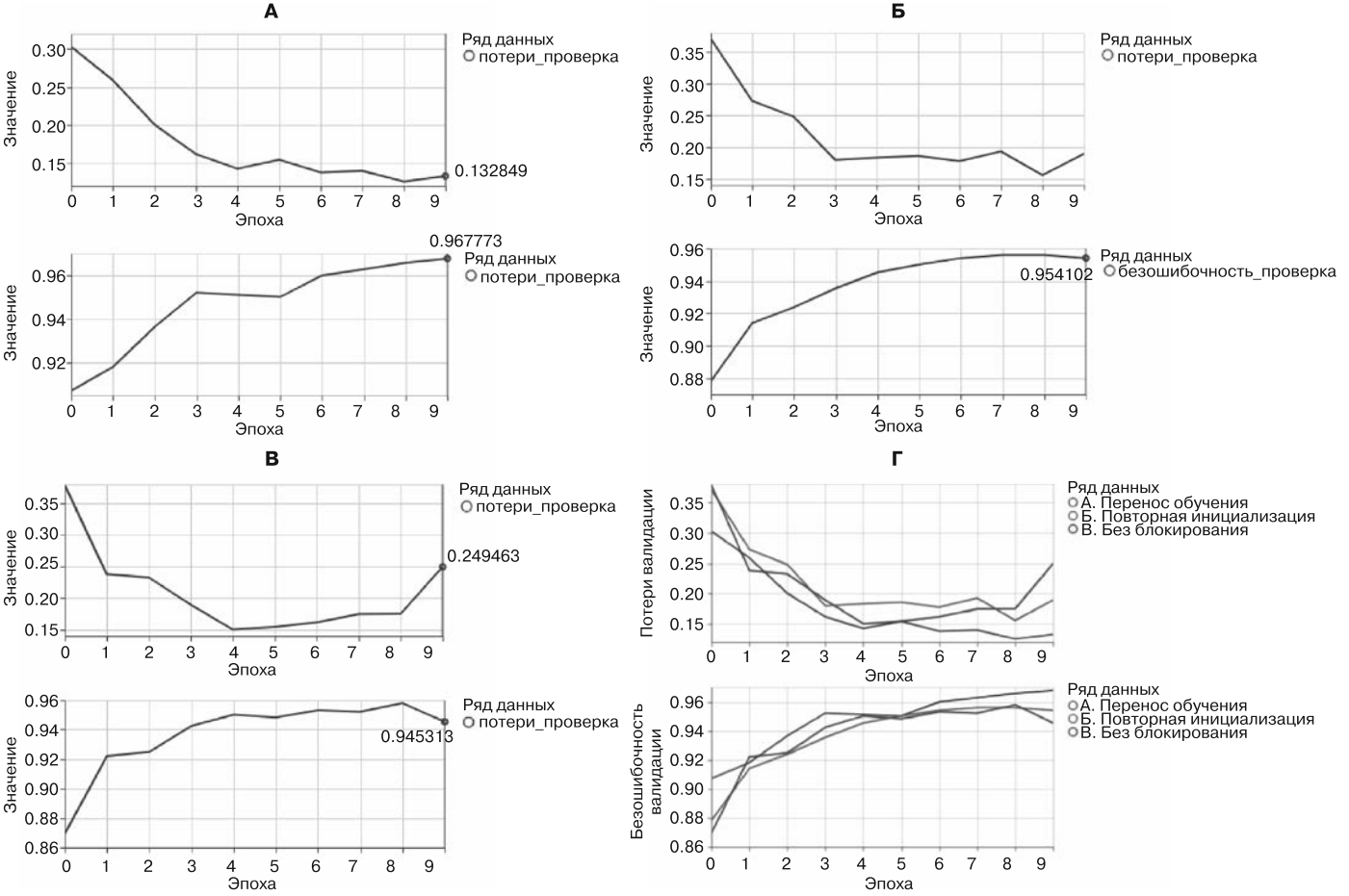


Рис. 5.3. Кривые потерь и безошибочности для переноса обучения на сверточной сети MNIST. Блок А: кривые при заблокированных первых семи слоях предобученной модели. Блок Б: кривые при повторной инициализации случайным образом значений всех весов модели. Блок В: кривые при отсутствии блокирования каких-либо слоев предобученной модели. Учтите, что оси у первых трех блоков отличаются. Блок Г: смешанный пример, на котором графики потерь и безошибочности из блоков А — В для удобства сравнения показаны на одних осях координат

Но что получится, если не блокировать веса слоев выделения признаков? Провести такой эксперимент вы можете с помощью опции Don't Freeze Feature Layers раскрывающегося меню Training Mode. Результат приведен в блоке В на рис. 5.3. Стоит упомянуть несколько его отличий от результатов из блока А.

- Без блокирования слоев выделения признаков начальное значение функции потерь оказывается значительно выше (например, после первой эпохи: 0,37 вместо 0,27), а безошибочность — ниже (0,87 вместо 0,91). Почему так? В начале обучения предобученной модели на новом наборе данных предсказания сначала содержат множество ошибок, поскольку для пяти новых цифр предобученные веса выдают, по сути, случайные предсказания. В результате значения функции потерь будут очень велики, а углы наклона ее графика — очень круты. Это приводит к большим значениям градиентов, вычисляемых на начальных этапах обучения, и, в свою очередь, ведет к сильным колебаниям значений всех весовых коэффициентов модели. А период сильных колебаний значений, через который проходят весовые коэффициенты всех слоев, приводит к более высоким начальным потерям, как видно из блока В. Поэтому при традиционном подходе к переносу обучения первые несколько слоев блокируются и тем самым «ограждаются» от этих больших начальных возмущений весов.
- В частности, из-за этих больших начальных возмущений итоговая безошибочность модели при подходе без блокирования (0,945, блок В) оказывается *не* выше, чем безошибочность обычного подхода переноса обучения с блокированием слоев (~0,968, блок А).
- Если ни один из слоев модели не заблокирован, обучение занимает намного больше времени. Например, на одном из наших ноутбуков обучение модели с заблокированными слоями выделения признаков заняло около 30 секунд, а обучение модели без каких-либо заблокированных слоев — примерно вдвое больше (60 секунд). На рис. 5.4 схематически показана причина этого явления. Заблокированные слои исключаются из уравнения во время обратного распространения ошибки, в результате чего каждый вызов `fit()` отрабатывается быстрее.

На рис. 5.4 путь обратного распространения ошибки показан черными стрелками, указывающими влево. В отсутствие заблокированных слоев все весовые коэффициенты модели (v_1–v_5) обновляются на каждом шаге обучения (каждом батче), а значит, участвуют в процессе обратного распространения ошибки (блок А). Обратите внимание, что признаки (x) и целевые переменные (y) в обратном распространении ошибки никогда не участвуют, поскольку их значения не нужно обновлять. Замораживание первых нескольких слоев модели приводит к исключению подмножества весов (v_1–v_3) из процесса обратного распространения ошибки (блок Б). Они становятся аналогами x и y и рассматриваются как константы, учитываемые при вычислении функции потерь. В результате объем необходимых для обратного распространения ошибки вычислений уменьшается, а процесс обучения ускоряется.

Приведенные различия подтверждают обоснованность подхода с блокированием слоев при переносе обучения: при нем слои выделения признаков из базовой

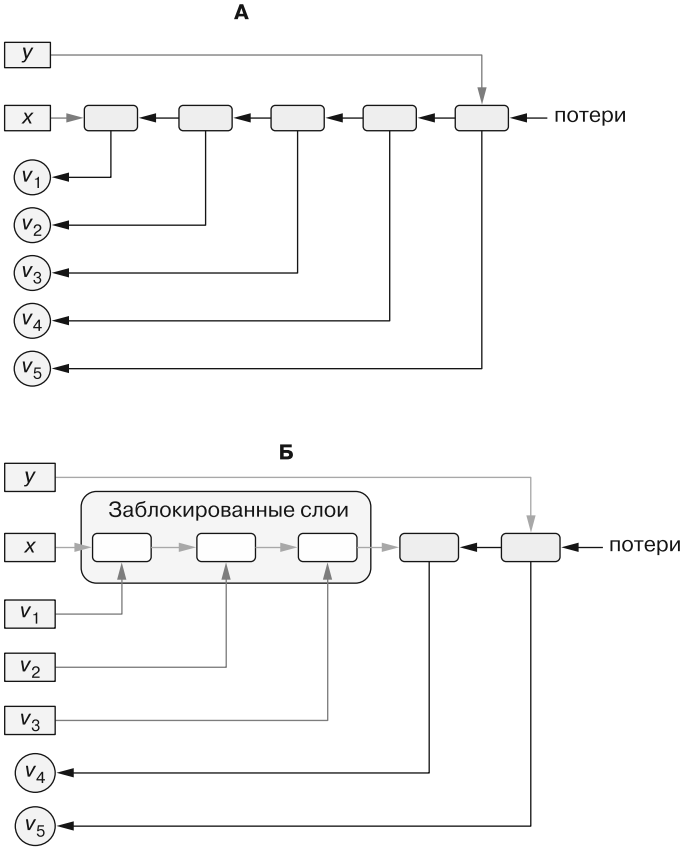


Рис. 5.4. Схематическое пояснение того, почему блокирование части слоев приводит к ускорению обучения

модели используются, но защищены от больших возмущений весов на начальных этапах нового обучения, благодаря чему можно достичь большей безошибочности за короткое время.

Два последних замечания, прежде чем перейдем к следующему разделу. Во-первых, при адаптации модели — процессе повторного обучения модели, чтобы она лучше работала на данных конкретного пользователя, — применяются очень схожие с показанными здесь методики, а именно, первые несколько слоев блокируются, а веса нескольких прочих слоев меняются в процессе обучения на данных, относящихся к конкретному пользователю. Хотя в этом разделе мы решаем задачу, связанную с данными не от другого пользователя, а просто с другими метками. Во-вторых, возможно, вам интересно, как проверить, остается ли тем же весовой коэффициент заблокированного слоя (слоев conv2d в данном случае) после вызова `fit()`. Проверить это несложно, мы оставим это вам в качестве задания (см. упражнение 2 в конце главы).

5.1.2. Перенос обучения при несовместимых формах выходных сигналов: создание новой модели на основе выходных сигналов базовой модели

В примере переноса обучения из предыдущего раздела форма выходного сигнала базовой модели была аналогична форме нового выходного сигнала. Во многих других сценариях переноса обучения это не так (рис. 5.5). Например, описанный выше подход не сработает, если нужно классифицировать *четыре* новые цифры с помощью базовой модели, обученной на пяти. Более распространенный сценарий: решить задачу классификации изображений по небольшому числу классов (случай Б на рис. 5.5) при наличии глубокой сверточной сети, обученной на наборе данных ImageNet, который предназначен для классификации изображений по 1000 выходных классов. Перед вами может стоять, например, задача бинарной классификации — содержит ли изображение человеческое лицо — или задача многоклассовой классификации всего по нескольким классам (вспомните пример в начале главы). В подобных случаях форма выходного сигнала базовой модели не подходит для новой задачи.

В некоторых случаях даже *тип* задачи машинного обучения отличается от типа задачи, для которой обучалась базовая модель. Например, можно решать задачу регрессии (предсказывать число, как в случае В на рис. 5.5) путем переноса обучения для базовой модели, обученной на задаче классификации. В разделе 5.2 вы увидите еще более интересный вариант использования переноса обучения — предсказание массива чисел, а не одного числа для обнаружения в изображениях объектов и определения их местоположения.

Подробнее разберем рис. 5.5. Случай А: форма выходного сигнала и функция активации новой модели совпадают с базовой моделью. Примером может служить перенос модели MNIST на новые цифры в подразделе 5.1.1. Случай Б: функции активации в новой и базовой моделях совпадают, поскольку совпадают типы исходной и новой задач (например, обе — задачи многоклассовой классификации). Однако формы выходных сигналов различаются (например, в новой задаче другое число классов). Примеры этого типа переноса обучения можно найти в подразделах 5.1.2 (управление компьютерной игрой в стиле Pac-Man™[1] с помощью веб-камеры) и 5.1.3 (распознавание нового множества слов устной речи). Случай В: новая задача — иного типа, чем исходная (например, регрессия вместо классификации). Примером может служить модель обнаружения объектов в изображениях, основанная на MobileNet.

Во всех описанных выше случаях форма желаемого выходного сигнала отличается от формы выходного сигнала базовой модели. Поэтому необходимо создать новую модель. Но раз уж речь идет о переносе обучения, создавать ее с нуля не нужно, можно использовать базовую. Мы продемонстрируем, как это сделать, в примере webcam-transfer-learning из репозитория tfjs-examples.

---

[1] Pac-Man — зарегистрированная торговая марка компании Bandai Namco Entertainment Inc.

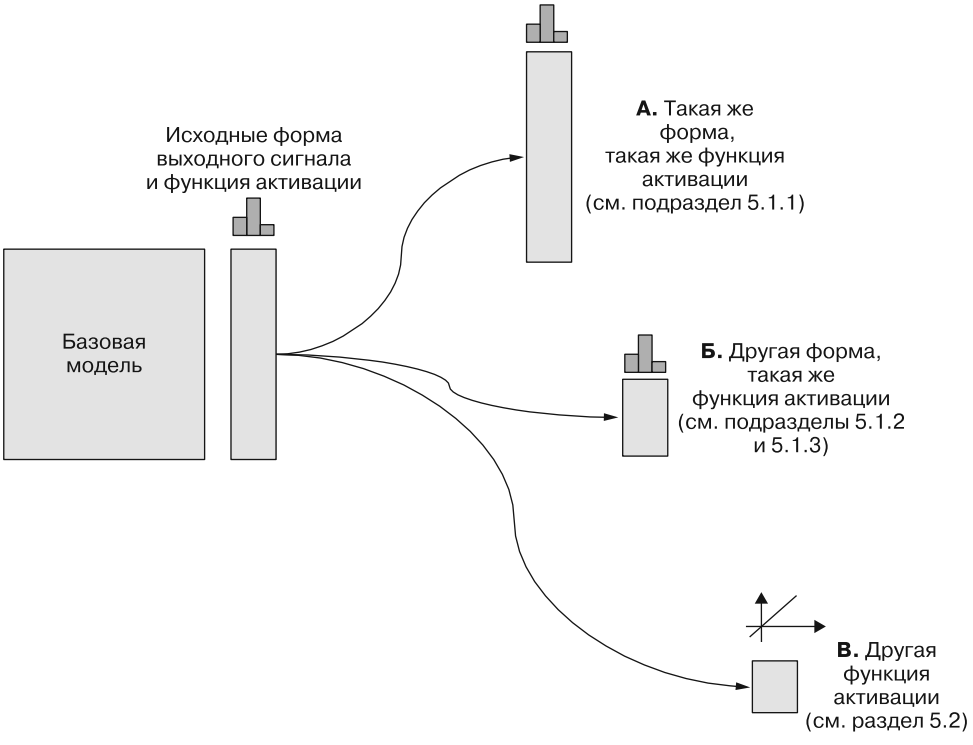


Рис. 5.5. Перенос обучения можно разделить на три разновидности, в зависимости от того, отличаются ли форма выходного сигнала и функция активации в новой и исходной моделях

Чтобы увидеть пример в действии, убедитесь, что у вашей машины есть фронтальная веб-камера — мы будем собирать с нее данные для переноса обучения. В настоящее время большинство ноутбуков и планшетов поставляются со встроенной фронтальной веб-камерой. Однако если вы работаете на стационарном компьютере, то вам, возможно, нужно будет достать веб-камеру и подключить ее к своей машине. Аналогично предыдущим примерам извлечь и запустить демонстрацию можно с помощью следующих команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/webcam-transfer-learning
```

Это забавное демонстрационное приложение превращает веб-камеру в игровую приставку благодаря применению переноса обучения к TensorFlow.js-реализации MobileNet и дает вам возможность с ее помощью играть в Pac-Man. Обсудим три необходимых для работы демонстрации шага: сбор данных, перенос обучения модели и игру.

Данные для переноса обучения собираются с веб-камеры. После запуска демонстрации в браузере вы увидите четыре черных квадрата в нижнем правом углу страницы. Они расположены аналогично четырем кнопкам перемещения в различных направлениях на игровой приставке Nintendo Family Computer и соответствуют

четырем классам, которые модель обучится распознавать в режиме реального времени. Эти четыре класса, в свою очередь, соответствуют четырем направлениям, в которых может перемещаться Пакман. При нажатии и удержании одной из них с веб-камеры будут собираться изображения с частотой 20–30 кадров в секунду. Число под квадратиком указывает, сколько именно изображений было собрано для данного направления на текущий момент времени.

Для наилучшего качества переноса обучения: 1) соберите по крайней мере 50 изображений каждого из классов и 2) немного покачайте головой и погримасничайте во время сбора данных для повышения разнообразия обучающих изображений, а значит, и устойчивости будущей модели, получаемой в результате переноса обучения. В этой демонстрации большинство людей поворачивают голову в четырех направлениях (вверх, вниз, влево и вправо; рис. 5.6), чтобы указать, куда Пакману нужно идти. Но можно использовать любые положения головы, любую мимику или даже жесты в качестве входных изображений, лишь бы входные данные внешне достаточно сильно отличались друг от друга.

Рис. 5.6. UI примера webcam-transfer-learning[1]

После сбора обучающих изображений нажмите кнопку Train Model, чтобы запустить процесс переноса обучения. Он займет лишь несколько секунд. Вы увидите, как значение выводимой на экране функции потерь падает до тех пор, пока не достигнет очень маленького положительного значения (например, 0,00010) и перестанет меняться. В этот момент модель переноса обучения уже обучена и с ее помощью можно играть. Для запуска игры просто нажмите кнопку Play и подождите, пока состояние игры не стабилизируется. После этого модель будет выводить результаты в режиме реального времени для потока получаемых с веб-камеры изображений. Для каждого кадра видео в нижнем правом углу UI будет выделяться ярко-желтым

[1] Созданием UI примера webcam-transfer-learning мы обязаны Джимбо Уилсону и Шену Картеру. Видеозапись этого развлекательного примера в действии вы можете найти по адресу https://youtu.be/YB-kfeNIPCE?t=941.

цветом класс-победитель (класс, которому модель присвоила максимальную вероятность). Кроме того, Пакман переместится в соответствующем направлении (если ему не помешает стена).

Для незнакомых с машинным обучением эта демонстрация выглядит каким-то колдовством, но в ее основе лежит простой алгоритм переноса обучения, использующий MobileNet для решения задачи четырехклассовой классификации. В алгоритме используется небольшое количество данных, собранных с помощью веб-камеры. Вы маркируете эти изображения с помощью нажатия и удержания клавиши во время сбора. Благодаря возможностям переноса обучения этот процесс не требует большого объема обучающих данных и не занимает много времени (может работать даже на смартфоне). Вот как, по существу, работает данная демонстрация. Если же вам интересны технические подробности, загляните далее вместе с нами в код TensorFlow.js.

Изучаем нюансы переноса обучения для веб-камеры

Базовую модель загружает код из листинга 5.3 (из файла `webcam-transfer-learning/index.js`). В частности, мы загружаем версию MobileNet, подходящую для работы в TensorFlow.js. В инфобоксе 5.1 описаны нюансы преобразования этой модели из формата библиотеки глубокого обучения Keras Python в формат TensorFlow.js. После загрузки модели мы используем метод `getLayer()` для получения ссылки на один из ее слоев. Метод `getLayer()` позволяет обозначать слои по названию (в данном случае `'conv_pw_13_relu'`). Наверное, вы помните и другой способ обращения к слоям модели из подраздела 2.4.2 — путем индексации атрибута `layers`, в котором все слои модели хранятся в виде JavaScript-массива. Этот подход удобен лишь тогда, когда модель состоит из относительно небольшого количества слоев. А наша MobileNet включает 93 слоя, так что подобный подход в ее случае ненадежен (что, если, например, в будущем в модель добавятся новые слои?). Надежнее будет воспользоваться подходом на основе названий и метода `getLayer()`, если допустить, что авторы MobileNet не станут менять названия ключевых слоев при выпуске новых версий модели.

Листинг 5.3. Загрузка MobileNet и создание на ее основе «усеченной» модели

```
async function loadTruncatedMobileNet() {
  const mobilenet = await tf.loadLayersModel(
    'https://storage.googleapis.com/' +
        'tfjs-models/tfjs/mobilenet_v1_0.25_224/model.json');

  const layer = mobilenet.getLayer(
      'conv_pw_13_relu');
  return tf.model({
    inputs: mobilenet.inputs,
    outputs: layer.output
  });
}
```

Мы постарались использовать в каталоге storage.google.com/tfjs-models постоянные, стабильные URL

Получаем один из промежуточных слоев MobileNet. Этот слой содержит удобные для нашей задачи классификации признаки

Создаем новую модель, такую же, как и MobileNet, за исключением того, что она оканчивается на слое 'conv_pw_13_relu', то есть несколько последних слоев (верхушка) усечены

ИНФОБОКС 5.1. Преобразование моделей из формата библиотеки Keras языка Python в формат TensorFlow.js

TensorFlow.js отличается высокой совместимостью с библиотекой Keras, одной из наиболее популярных библиотек глубокого обучения Python. Одно из преимуществ этой совместимости — возможность использовать многие так называемые приложения из Keras, представляющие собой наборы предобученных глубоких сверточных сетей (см. https://keras.io/applications/). Создатели Keras старательно обучили эти сверточные сети на больших наборах данных (наподобие ImageNet) и открыли доступ к ним через библиотеку Keras для свободного переиспользования. В том числе для вывода и переноса обучения, подобно тому как мы делаем в книге. Импорт приложения занимает всего одну строку кода при использовании Keras в Python. Благодаря вышеупомянутой широкой совместимости пользователи TensorFlow.js также с легкостью могут работать с этими приложениями. Вот что для этого требуется.

1. Убедитесь, что у вас установлен пакет Python `tensorflowjs`. Проще всего установить его с помощью команды `pip`:

   ```
   pip install tensorflowjs
   ```

2. Выполните следующий код из файла исходного кода Python или в интерактивной среде REPL наподобие ipython:

   ```
   import keras
   import tensorflowjs as tfjs
   model = keras.applications.mobilenet.MobileNet(alpha=0.25)
   tfjs.converters.save_keras_model(model, '/tmp/mobilnet_0.25')
   ```

Первые две строки импортируют требуемые модули `keras` и `tensorflowjs`. Третья строка загружает MobileNet в объект языка Python (`model`). При желании можно вывести текстовую сводку модели практически так же, как мы выводили сводки топологии моделей TensorFlow.js: с помощью `model.summary()`. Как видите, форма последнего слоя модели (выходного слоя) действительно `(None, 1000)` (эквивалентно `[null, 1000]` в JavaScript), отражая задачу 1000-классовой классификации ImageNet, на которой обучалась модель MobileNet. Указание в вызове этого конструктора именованного аргумента `alpha=0.25` позволяет выбрать меньшую по размеру версию MobileNet. Можно выбрать и большие значения `alpha` (например, `0.75` или `1`), и будет работать тот же самый код преобразования.

Последняя строка в предыдущем фрагменте кода сохраняет модель в указанный каталог на диске с помощью метода из модуля tensorflowjs. После завершения выполнения этой строки в каталоге `/tmp/mobilnet_0.25` появится новый подкаталог, содержимое которого будет примерно таким:

```
group1-shard1of6
    group1-shard2of6
    ...
    group1-shard6of6
    model.js1on
```

Именно такой формат мы видели в подразделе 4.3.3, когда сохраняли обученную модель TensorFlow.js на диск с помощью ее метода `save()` в версии TensorFlow.js для Node.js. Следовательно, с точки зрения основанной на TensorFlow.js программы,

загружающей эту преобразованную модель с диска, сохраненный формат идентичен модели, созданной и обученной в TensorFlow.js: можно просто вызвать метод `tf.loadLayersModel()`, передав ему путь к файлу `model.json` (либо в браузере, либо в Node.js). Именно это и происходит в листинге 5.3.

Загруженная модель MobileNet готова к выполнению задачи ML, для которой она изначально была обучена, — классификации входных изображений по 1000 классов набора данных ImageNet. Учтите, что этот конкретный набор данных в значительной степени ориентирован на изображения животных, особенно различные породы кошек и собак (вероятно, это связано с большой распространенностью подобных изображений в Интернете). Этот сценарий использования иллюстрирует пример MobileNet из репозитория tfjs-example (https://github.com/tensorflow/tfjs-examples/tree/master/mobilenet). Впрочем, в этой главе нас не интересует подобное непосредственное применение MobileNet; мы хотим воспользоваться загруженной моделью для переноса обучения.

Приведенный выше метод `tfjs.converters.save_keras_model()` умеет преобразовывать и сохранять не только модель MobileNet, но и другие приложения Keras, например DenseNet и NasNet. В упражнении 3 в конце главы вам предстоит попытаться преобразовать в формат TensorFlow.js другое приложение Keras (MobileNetV2) и загрузить его в браузере. Более того, стоит отметить, что метод `tfjs.converters.save_keras_model()` можно использовать вообще для любых объектов моделей, созданных или обученных в Keras, а не только моделей из модуля `keras.applications`.

Что же мы будем делать со слоем `conv_pw_13_relu` после получения на него ссылки? Создадим новую модель, включающую слои исходной модели MobileNet, начиная с ее первого (входного) слоя и до слоя `conv_pw_13_relu`. В книге вы первый раз сталкиваетесь с подобным стилем конструирования модели, так что не помешает пояснить его идею подробнее. Для этого нам понадобится ввести понятие *символического тензора* (symbolic tensor).

Создание моделей из символических тензоров

Вы уже знакомы с понятием тензора. `Tensor` — базовый тип данных (сокращенно *dtype*) в TensorFlow.js. Объект тензора содержит конкретные числовые значения заданной формы и типа, хранимые в текстурах WebGL (если речь идет о браузере с поддержкой WebGL) или памяти CPU/GPU (в Node.js). `SymbolicTensor` — отдельный важный класс библиотеки TensorFlow.js. Символический тензор не содержит конкретных значений, только описывает их форму и dtype. Его можно рассматривать как слот (заполнитель), в который позднее можно вставить фактический тензор при условии совместимости формы и типа значения тензора. В TensorFlow.js объект слоя или модели принимает один или несколько входных сигналов (до сих пор мы сталкивались только со случаями одного входного сигнала), представленных в виде одного или нескольких символических тензоров.

Воспользуемся аналогией, чтобы лучше понять, что такое символические тензоры. Представьте себе функцию языка программирования, например Java или TypeScript (или любого другого знакомого вам статически типизированного языка

программирования). Она принимает один или несколько входных аргументов. У каждого аргумента есть тип, определяющий, какие переменные через него можно передать. Однако *сам* этот аргумент не содержит никаких конкретных значений, он просто заполнитель. Символический тензор аналогичен аргументу функции: он просто задает, тензоры какого вида (сочетания формы[1] и dtype) могут использоваться в этом слоте. У функций в статически типизированных языках программирования есть и возвращаемый тип данных, который можно сравнить с выходным символическим тензором объекта модели или слоя. Он представляет собой своего рода шаблон формы и dtype фактических значений выходного тензора объекта, принадлежащего модели или слою.

Два важных атрибута объекта модели в TensorFlow.js — ее входные и выходные сигналы. Каждый из них представляет собой массив символических тензоров. Длина обоих массивов у модели с одним входным и одним выходным сигналом равна 1. Аналогичным образом, у объекта слоя два атрибута: входные и выходные сигналы, оба — символические тензоры. Символические тензоры можно использовать при создании новой модели. Это новый для нас способ создания моделей в TensorFlow.js, отличающийся от описанного ранее подхода, а именно создания последовательных моделей с помощью метода `tf.sequential()` и дальнейших вызовов метода `add()`. В этом новом для нас подходе используется функция `tf.model()`, принимающая на входе объект конфигурации, включающий два обязательных поля: `inputs` и `outputs`. Поле `inputs`, как и `outputs`, должно представлять собой символический тензор (либо массив символических тензоров). Следовательно, можно получить символические тензоры из исходной модели MobileNet и передать их в метод `tf.model()`. Результатом вызова этого метода будет новая модель, частично состоящая из старой модели MobileNet.

Этот процесс схематически изображен на рис. 5.7 (учтите, что на рисунке число слоев уменьшено по сравнению с настоящей моделью MobileNet ради упрощения схемы). Важно отдавать себе отчет, что получаемые из исходной модели и передаваемые методу `tf.model()` символические тензоры — *не* изолированные объекты. Они содержат информацию о слоях, к которым относятся, и о том, как эти слои соединяются друг с другом. Для тех из наших читателей, кто знаком с теорией графов в структурах данных, исходная модель представляет собой граф из символических тензоров, где слои играют роль соединяющих их ребер. Задавая входные и выходные сигналы новой модели в виде символических тензоров для исходной модели, мы выделяем подграф исходного графа MobileNet. Этот подграф, становясь новой моделью, содержит первые несколько (а именно, первые 87) слоев MobileNet, а последние шесть слоев отбрасываются. Последние несколько слоев глубокой сверточной сети иногда называют ее *верхушкой* (head). А вызовом `tf.model()` мы производим так называемое *усечение* (truncating) модели. В усеченной MobileNet остаются слои выделения признаков, а верхушка отбрасывается. Почему верхушка содержит

[1] Разница между формой обычного и формой символического тензора в том, что у первого всегда полностью заданы измерения (например, [8, 32, 20]), в то время как измерения второго определены не полностью (например, [null, null, 20]). Вы уже могли видеть подобное в столбце Output shape (Форма выходного сигнала) сводок моделей.

именно *шесть* слоев? Потому, что эти слои связаны сугубо с задачей 1000-классовой классификации, для которой изначально обучалась модель MobileNet. В ожидающей нас задаче четырехклассовой классификации они ничем не помогут.

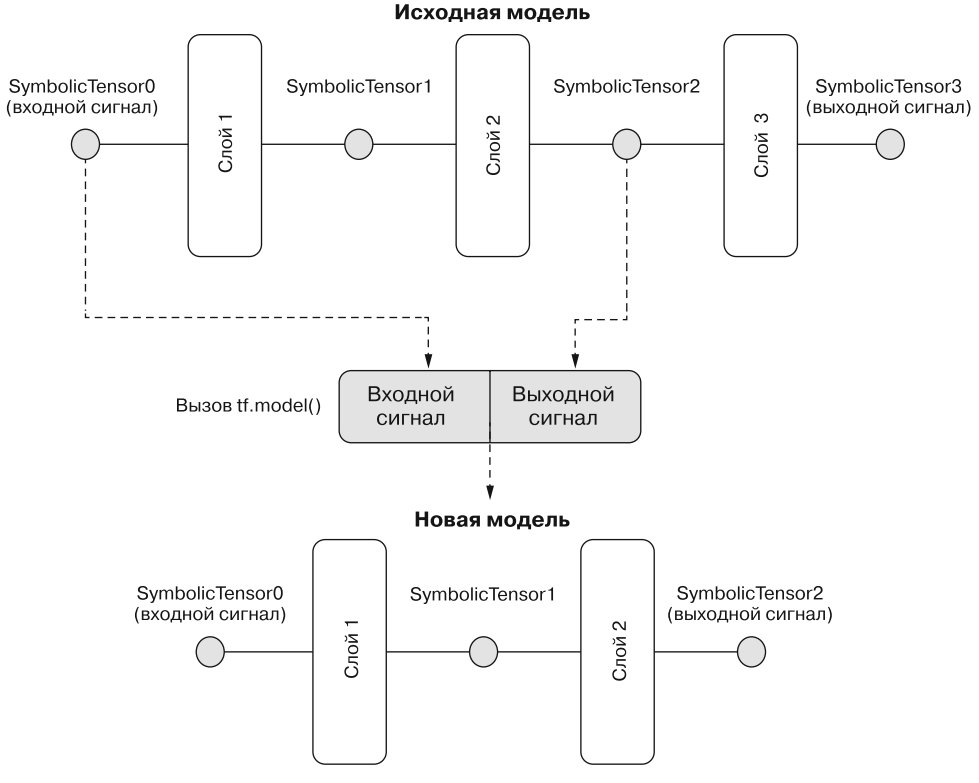


Рис. 5.7. Схематическая иллюстрация создания новой (усеченной) модели из MobileNet. Соответствующий код — см. вызов tf.model() в листинге 5.3

Рассмотрим подробнее рис. 5.7. У всех слоев есть входной сигнал и выходной сигнал, представляющие собой экземпляры класса `SymbolicTensor`. В исходной модели `SymbolicTensor0` представляет собой входной сигнал первого слоя (он же входной сигнал модели в целом). Он служит входным символическим тензором новой модели. Кроме того, выходной символический тензор промежуточного слоя (эквивалентного conv_pw_13_relu) используется в качестве выходного тензора новой модели. Таким образом, получается модель, состоящая из двух первых слоев исходной модели, показанная в нижней части схемы. Последний (выходной) слой исходной модели, иногда называемый верхушкой модели, мы отбрасываем. Именно по этой причине подобный подход иногда называют *усечением* модели. Учтите, что ради ясности модели на этой схеме включают лишь небольшое число слоев. В коде из листинга 5.3 модель состоит из гораздо большего числа слоев (93), чем приведенная на схеме.

Перенос обучения на основе вложений

Выходным сигналом усеченной MobileNet служит функция активации промежуточного слоя исходной MobileNet[1]. Но чем для нас полезны активации промежуточных слоев из MobileNet? Ответ заключается в функции, обрабатывающей события нажатия и удержания каждого из четырех черных квадратов (листинг 5.4). Каждый раз при получении входного изображения от веб-камеры (с помощью метода `capture()`) вызывается метод `predict()` усеченной MobileNet, а выходной сигнал сохраняется в объекте `controllerDataset`, чтобы его можно было позднее использовать при переносе обучения.

Но как интерпретировать выходной сигнал усеченной MobileNet? Для каждого входного изображения он представляет собой тензор формы `[1, 7, 7, 256]`. Это не вероятности для задачи классификации и не предсказанные значения для задачи регрессии, а представление входного изображения в многомерном ($7 \times 7 \times 256$, то есть примерно 12 500 измерений) пространстве. И хотя число измерений в этом пространстве очень велико, по сравнению с исходным изображением ($224 \times 224 \times 3 \approx$ $\approx$ 150 000 измерений) оно относительно низкой размерности. Так что можно считать выходной сигнал усеченной MobileNet эффективным представлением изображения. Подобное представление пониженной размерности для входных данных часто называют *вложением* (embedding). В основе нашего переноса обучения лежат вложения четырех множеств изображений, полученных с веб-камеры.

Листинг 5.4. Получение вложений изображений с помощью усеченной MobileNet

```
ui.setExampleHandler(label => {
  tf.tidy(() => {
    const img = webcam.capture();
    controllerDataset.addExample(
        truncatedMobileNet.predict(img),
        label);
    ui.drawThumb(img, label);
  });
});
```

Очищаем промежуточные тензоры, в частности img, с помощью tf.tidy(). См. руководство по управлению памятью в браузере TensorFlow.js (раздел Б.3)

Получаем внутреннюю функцию активации MobileNet для нашего входного изображения

Теперь у нас есть способ получить вложения изображений веб-камеры. Но как с их помощью предсказать, какому направлению соответствует конкретное изображение? Для этого нам понадобится новая модель, входным сигналом которой служат эти вложения, а выходным сигналом — значения вероятностей для четырех классов направлений. Для создания подобной модели можно взять код из листинга 5.5 (из файла `index.js`).

Новая модель, созданная в листинге 5.5, намного меньше усеченной MobileNet. Она состоит лишь из трех слоев.

- Входной слой — слой схлопывания, преобразующий трехмерные вложения из усеченной модели в одномерный тензор, подходящий в качестве входных данных

[1] Часто спрашивают, как получить значения функций активации промежуточных слоев моделей TensorFlow.js. Приведенная здесь методика может стать ответом на этот вопрос.

последующим плотным слоям. В главе 4 мы уже использовали аналогичным образом слои схлопывания в сверточных сетях MNIST. `inputShape` соответствует форме выходного сигнала исходной усеченной MobileNet (без измерения батчей), поскольку на вход новой модели будут подаваться вложения из усеченной MobileNet.

- Второй слой — скрытый слой. Он скрыт, поскольку не является ни входным, ни выходным слоем модели, а расположен между двумя другими слоями для повышения разрешающих возможностей модели, что очень напоминает MLP из главы 3. Это скрытый плотный слой с функцией активации ReLU. Как вы помните, в пункте «Избегаем нагромождения слоев без нелинейностей» главы 3 мы обсуждали важность использования нелинейных функций активации для подобных скрытых слоев.
- Третий слой — завершающий (выходной) слой новой модели. Он включает многомерную логистическую функцию активации, подходящую для решения нашей задачи многоклассовой (то есть четырехклассовой: по одному классу для каждого направления движения Пакмана) классификации.

Листинг 5.5. Предсказание направлений движения для игровой приставки с помощью вложений изображений

```
model = tf.sequential({
  layers: [
    tf.layers.flatten({
      inputShape: truncatedMobileNet.outputs[0].shape.slice(1)
    }),
    tf.layers.dense({
      units: ui.getDenseUnits(),
      activation: 'relu',
      kernelInitializer: 'varianceScaling',
      useBias: true
    }),
    tf.layers.dense({
      units: NUM_CLASSES,
      kernelInitializer: 'varianceScaling',
      useBias: false,
      activation: 'softmax'
    })
  ]
});
```

Первый (скрытый) плотный слой с нелинейной (ReLU) функцией активации

Схлопывает вложение с формой [7, 7, 256] из усеченной MobileNet. Операция slice(1) отбрасывает включенное в форму выходного сигнала, но не требуемое атрибутом inputShape фабричного метода слоя первое измерение (измерение батчей), чтобы этот выходной сигнал можно было использовать для плотного слоя

Число нейронов последнего слоя должно соответствовать числу предсказываемых классов

Таким образом, мы, по существу, построили MLP поверх слоев выделения признаков MobileNet. Этот многослойный перцептрон можно рассматривать как новую верхушку MobileNet, несмотря на то что средство выделения признаков (усеченная MobileNet) и верхушка в данном случае представляют собой две отдельные модели (рис. 5.8). В результате подобной схемы из двух моделей обучать новую верхушку непосредственно на тензорах изображений (формы `[numExamples, 224, 224, 3]`) невозможно. Следует обучать ее на вложениях изображений — выходном сигнале

усеченной MobileNet. К счастью, у нас уже есть эти тензоры вложений (см. листинг 5.4). Все, что нам осталось сделать для обучения новой верхушки, — вызвать ее метод `fit()`, передав ему тензоры вложений. Выполняющий это код расположен внутри функции `train()` в файле `index.js` и несложен, и мы не станем на нем останавливаться.

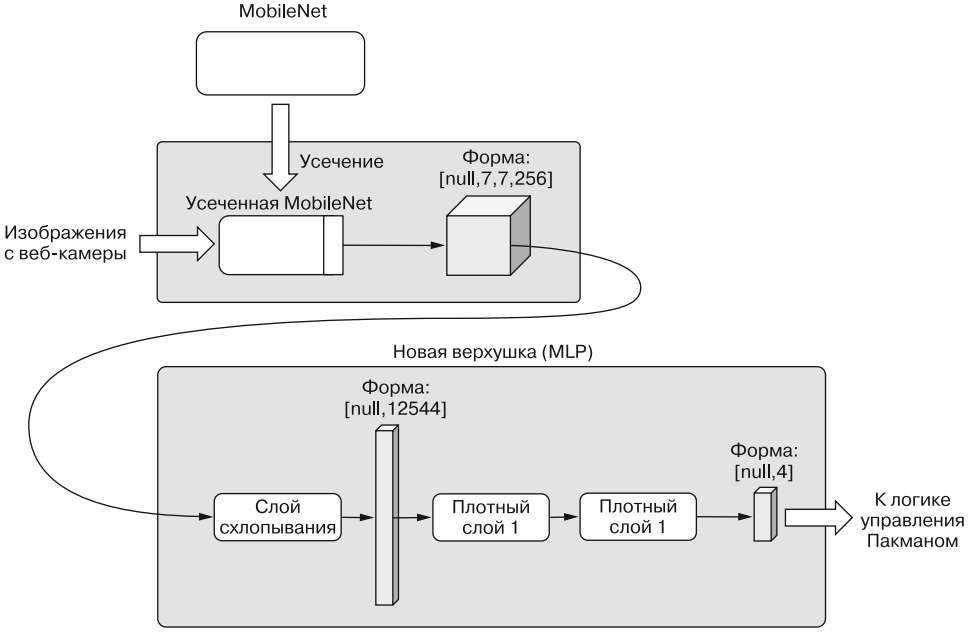


Рис. 5.8. Схематическое описание алгоритма машинного обучения, лежащего в основе примера webcam-transfer-learning

По завершении переноса обучения усеченная модель и новая верхушка вместе позволяют нам получить оценки вероятностей для входных изображений с вебкамеры. Соответствующий код можно найти в функции `predict()` в файле `index.js`, приведенной в листинге 5.6. В частности, в нем выполняются два вызова `predict()`, первый из которых преобразует тензор изображения в его вложение с помощью усеченной MobileNet, а второй преобразует вложение в оценки вероятностей четырех направлений с помощью новой верхушки, обученной путем переноса обучения. Дальнейший код из листинга 5.6 служит для получения индекса-победителя (индекса, соответствующего максимальной оценке вероятности для четырех направлений) и направления соответствующим образом движения Пакмана с обновлением состояния UI. Как и в предыдущих примерах, мы не станем описывать часть примера, относящуюся к UI, поскольку для нашего обсуждения алгоритмов машинного обучения она не столь важна. Можете изучить код UI и поэкспериментировать с ним, воспользовавшись кодом из листинга 5.6.

Листинг 5.6. Получаем предсказание для входного изображения с веб-камеры с помощью переноса обучения

```
async function predict() {
  ui.isPredicting();
  while (isPredicting) {
    const predictedClass = tf.tidy(() => {
      const img = webcam.capture();

      const embedding =
          truncatedMobileNet.predict(img);
      const predictions = model.predict(activation);
      return predictions.as1D().argMax();
    });

    const classId = (await predictedClass.data())[0];
    predictedClass.dispose();
    ui.predictClass(classId);
    await tf.nextFrame();
  }
  ui.donePredicting();
}
```

Захват изображения с веб-камеры

Получаем вложение от усеченной MobileNet

Преобразуем это вложение в оценки вероятностей четырех направлений с помощью новой верхушки модели

Получаем индекс максимальной оценки вероятности

Загружаем этот индекс из GPU в CPU

Обновляем UI в соответствии с «выигравшим» направлением: перемещаем Пакмана и обновляем прочие состояния UI, например подсвечиваем соответствующую «кнопку игровой приставки»

На этом мы завершаем обсуждение части примера webcam-transfer-learning, связанной с алгоритмом переноса обучения. Любопытный нюанс метода, применявшегося в этом примере: в процессе обучения и вывода участвуют два отдельных объекта моделей. Это удобно для иллюстрации того, как получить вложения из промежуточных слоев предобученной модели. Еще одно преимущество данного подхода: доступность вложений, благодаря чему упрощается применение напрямую использующих их методик ML. Один из примеров подобных методик — *метод k-ближайших соседей* (k-nearest neighbors, kNN, обсуждается в инфобоксе 5.2). Впрочем, возможность прямого доступа к вложениям по некоторым причинам может рассматриваться и как недостаток.

- Некоторое усложнение кода. Например, выполнение вывода для каждого изображения требует двух вызовов `predict()`.
- Пусть нам нужно сохранить модели для использования в будущем или для преобразования в формат какой-либо другой библиотеки, не TensorFlow.js. В этом случае необходимо сохранять усеченную модель и новую верхушку модели отдельно, как два отдельных артефакта.
- В особых случаях перенос обучения включает обратное распространение ошибки по определенным частям базовой модели (например, по первым нескольким слоям усеченной MobileNet). Это будет невозможно, если базовая модель и верхушка представляют собой два отдельных объекта.

В следующем подразделе мы покажем, как можно преодолеть эти ограничения путем формирования единого объекта модели для переноса обучения. Такая модель будет сквозной в том смысле, что будет преобразовывать входные данные в исходном формате в желаемый итоговый выходной сигнал.

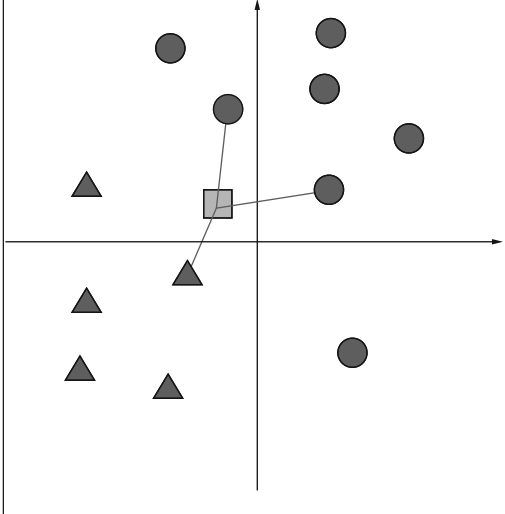
ИНФОБОКС 5.2. Классификация методом k-ближайших соседей на основе вложений

Решать задачи классификации в машинном обучении можно и без помощи нейронных сетей. Один из самых известных подобных подходов — алгоритм k-ближайших соседей (kNN). В отличие от нейронных сетей, в алгоритме kNN отсутствует этап обучения и он намного понятнее.

Работу метода классификации kNN можно описать в нескольких фразах.

1. Выбираем положительное целое число *k* (например, 3).
2. Собираем определенное число эталонных примеров данных с метками истинных классов. Обычно число эталонных примеров хотя бы в несколько раз превышает *k*. Примеры представляются в виде наборов вещественных чисел — *векторов*. Этот шаг схож со сбором обучающих примеров данных при использовании нейронных сетей.
3. Для предсказания класса нового входного сигнала вычисляются расстояния между векторным представлением этого сигнала и каждого из эталонных примеров данных. Затем расстояния сортируются, чтобы найти *k* эталонных примеров, ближайших к входному сигналу в векторном пространстве. Они и называются *k* ближайшими соседями входного сигнала (в честь чего алгоритм и получил свое название).
4. Выбираем класс, чаще всего встречающийся среди классов *k* ближайших соседей, в качестве предсказания для входного сигнала. Другими словами, *k* ближайших соседей выбирают предсказываемый класс путем своеобразного голосования.

Пример использования этого алгоритма показан на следующем рисунке.



Пример kNN-классификации в двумерном пространстве вложений. В данном случае k = 3, а классов два (треугольники и круги). Имеется пять эталонных примеров данных для класса «треугольник» и семь — для класса «круг». Три ближайших соседа входного сигнала показаны отрезками прямых. А поскольку два из трех ближайших соседей — круги, то и предсказан будет класс «круг»

Как вы можете видеть из предыдущего описания, одно из ключевых требований алгоритма kNN — представление всех входных примеров данных в виде векторов. Полученные из усеченной MobileNet вложения и им подобные отлично подходят для таких векторных представлений по двум причинам. Во-первых, их размерность обычно ниже, чем у исходных входных данных, а потому для вычисления расстояний требуется меньше вычислений и места для хранения. Во-вторых, вложения обычно захватывают самые важные признаки входных данных (например, существенные геометрические признаки на изображениях; см. рис. 4.5), игнорируя менее важные (яркость и размер), благодаря обучению на большом наборе данных для классификации. В некоторых случаях с помощью вложений можно получить векторные представления объектов, изначально не представленных в числовом виде (например, с помощью вложений слов, см. главу 9).

В отличие от подхода нейронных сетей, метод kNN не требует обучения. При небольшом числе эталонных примеров данных и небольшой размерности входного сигнала метод kNN с вычислительной точки зрения эффективнее обучения нейронной сети и выполнения с ее помощью вывода.

Впрочем, вывод с использованием kNN плохо масштабируется при росте объемов данных. В частности, при N эталонных примерах данных классификатору kNN для предсказания класса каждого входного сигнала необходимо вычислить N расстояний[1]. Количество необходимых вычислений при больших N становится неподъемным. И напротив, объемы обучающих данных не влияют на вывод с помощью нейронной сети. По завершении обучения сети уже неважно, сколько примеров данных участвовало в работе. Объемы вычислений, требуемых для прямого прохода, зависят только от топологии сети.

Если вы хотели бы попробовать kNN для своих приложений, посмотрите на надстройку для TensorFlow.js — библиотеку kNN с WebGL-ускорением: http://mng.bz/2Jp8.

5.1.3. Извлекаем максимум пользы из переноса обучения благодаря тонкой настройке: пример обработки аудиоданных

В предыдущих разделах в примерах переноса обучения речь шла об обработке визуальных данных. В этом же примере мы продемонстрируем перенос обучения для аудиоданных, представленных в виде изображений спектрограмм. Как вы помните, в разделе 4.4 мы описывали сверточную сеть для распознавания речевых команд (отдельных, коротких слов). Созданное тогда средство распознавания speech-command могло распознавать лишь 18 различных слов (таких как *one*, *two*, *up* и *down*). А что, если нужно обучить его распознавать и другие слова? Например, пусть для вашего

[1] Впрочем, исследователи прилагают немалые усилия для разработки алгоритмов приближенной реализации kNN, быстрее работающих и лучше масштабирующихся: *Yona G.* Fast Near-Duplicate Image Search Using Locality Sensitive Hashing // Towards Data Science, 5 May 2018. http://mng.bz/1wm1.

приложения требуется, чтобы пользователь говорил конкретные слова, например *red* или *blue*, либо какие-то слова, выбранные самими пользователями; либо приложение будет предназначено для носителей других языков, помимо английского. Это классический пример переноса обучения: можно *попытаться* обучить модель с нуля на небольшом количестве данных, но благодаря использованию предобученной модели в качестве базовой можно сэкономить немало времени и вычислительных ресурсов при более высокой степени безошибочности.

Как выполнить перенос обучения в примере приложения speech-command

Прежде чем описать схему работы переноса обучения, расскажем вам, как использовать возможности переноса обучения через UI. Для этого убедитесь, что к вашей машине подключено устройство ввода звука (микрофон), а уровень громкость в системных настройках ненулевой. Чтобы скачать и запустить код демонстрации, выполните следующие команды (та же самая процедура, что и в подразделе 4.4.1):

```
git clone https://github.com/tensorflow/tfjs-models.git
cd tfjs-models/speech-commands
yarn && yarn publish-local
cd demo
yarn && yarn link-local && yarn watch
```

После запуска пользовательского интерфейса ответьте «да» на запрос браузера, разрешив доступ к микрофону. Рисунок 5.9 демонстрирует пример снимка экрана демонстрации. При запуске страница демонстрации автоматически загружает из Интернета предобученную модель speech-command с помощью метода `tf.loadLayersModel()`, параметр которого указывает URL по протоколу HTTPS. После загрузки модели станут доступны кнопки Start и Enter Transfer Words. Если нажать кнопку Start, демонстрация начнет работать в режиме вывода, в котором будет распознано 18 основных слов (как показано на экране) в непрерывном режиме. При каждом обнаружении слова на экране подсвечивается соответствующее окошко для слова. Если же нажать кнопку Enter Transfer Words, на экране появится несколько дополнительных кнопок, созданных на основе списка разделенных запятыми слов из поля текстового ввода справа. По умолчанию это слова *noise*, *red* и *green*. Именно их будет обучаться распознавать модель, полученная с помощью переноса обучения. Но вы можете свободно менять содержимое окна ввода, если хотите обучить перенесенную модель распознавать другие слова, главное, сохранить *noise*. Элемент *noise* — особенный, для него необходимо собрать образцы фонового шума, то есть примеры данных, не содержащие никаких звуков речи. Благодаря ему перенесенная модель обучается отличать моменты, когда произносятся слова, от промежутков тишины (когда присутствует только фоновый шум). При нажатии этих кнопок демонстрация записывает с микрофона звуковые фрагменты длительностью одна секунда и отображает их спектрограммы рядом с соответствующими кнопками. Числа на кнопках отражают количество собранных на текущий момент примеров данных для соответствующего слова.

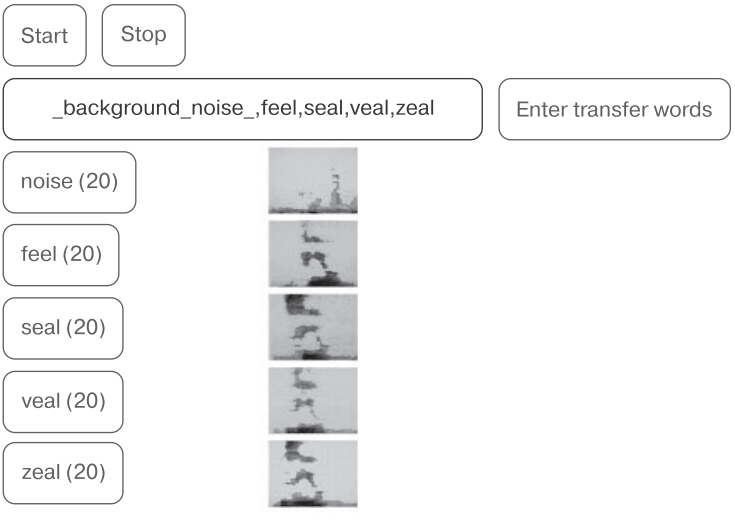


Рис. 5.9. Один из снимков экрана переноса обучения для примера speech-command. В данном случае пользователь ввел некий набор слов для переноса обучения: feel, seal, veal и zeal, помимо обязательного элемента noise. Кроме того, пользователь собрал по 20 примеров данных для каждой категории слов и шума

Как это обычно бывает в задачах машинного обучения, чем больше данных удалось собрать (чем больше позволили имеющиеся ресурсы и время), тем лучше оказывается обученная модель. Наш пример приложения требует по крайней мере восемь примеров данных для каждого слова. Если вы сами не хотите или не можете собрать примеры аудиоданных, можете скачать готовый набор данных по адресу http://mng.bz/POGY (размер файла — 9 Мбайт) и загрузить его, нажав кнопку Upload в разделе Dataset IO пользовательского интерфейса.

После подготовки набора данных путем загрузки файла или на основе собранных примеров станет доступна кнопка Start Transfer Learning. Можете ее нажать, чтобы запустить перенос обучения модели. Приложение выполняет разбиение собранных спектрограмм звука в отношении 3:1, так что случайно выбранные 75 % из них используются для обучения, а оставшиеся 25 % — для проверки[1]. Приложение отображает потери и показатель безошибочности как на обучающем, так и проверочном наборе данных по мере хода переноса обучения. По завершении обучения нажмите кнопку Start для запуска непрерывного распознавания новых слов, в ходе чего можно опытным путем оценить безошибочность перенесенной модели.

Желательно поэкспериментировать с различными наборами слов и посмотреть, какой степени безошибочности можно достичь после переноса обучения для них. В наборе слов по умолчанию, *red* и *green*, слова сильно отличаются друг от друга

[1] Именно поэтому для демонстрационного примера необходимо собрать по крайней мере по восемь примеров данных для каждого слова. При меньшем количестве число примеров в проверочном наборе данных для каждого слова будет слишком маленьким, что приведет к потенциально недостоверным оценкам потерь и безошибочности.

в смысле фонематического содержания. Гласные в них также звучат достаточно поразному (*e* и *ee*), как и завершающие согласные (*d* и *n*). Следовательно, безошибочность на проверочном наборе данных в конце переноса обучения должна оказаться почти идеальной, конечно, если число примеров данных, собранных для каждого слова, достаточно велико (скажем, ≥ 8), а, кроме того, число эпох не слишком мало (что приводит к недообучению) и не слишком велико (что приводит к переобучению; см. главу 8).

Чтобы усложнить задачу переноса обучения для модели, возьмите набор, состоящий из: 1) хуже различимых слов и 2) большего количества слов. Пример такого набора приведен на рис. 5.9. В нем используется набор из четырех схожих по звучанию слов: *feel*, *seal*, *veal* и *zeal*. Гласные и завершающие согласные у этих слов совпадают, а начальные согласные звучат очень похоже. Их может легко спутать даже человек — невнимательный или слышащий их по телефонной линии с плохим качеством связи. Кривая безошибочности справа внизу на рисунке демонстрирует, что модели непросто достичь безошибочности более 90 %. Для этого начальный этап переноса обучения должен сопровождаться дополнительным этапом *тонкой настройки* (fine-tuning) — одного из остроумных приемов переноса обучения.

Углубляемся в нюансы тонкой настройки при переносе обучения

Тонкая настройка — методика, с помощью которой можно достичь уровня безошибочности, недостижимого за счет простого обучения новой верхушки перенесенной модели. В этом разделе вы во всех подробностях узнаете, как происходит тонкая настройка. При этом вам придется «переварить» несколько технических нюансов. Но более глубокое понимание переноса обучения, а значит, и соответствующей реализации TensorFlow.js стоит затраченных усилий.

Формирование отдельной модели для переноса обучения

Во-первых, разберемся, как в приложении для распознавания речи создается модель для переноса обучения. В коде из листинга 5.7 (из файла `speech-commands/src/browser_fft_recognizer.ts`) создается модель из базовой модели speech-command (с которой мы познакомились в подразделе 4.4.1). Сначала мы находим предпоследний плотный слой модели и получаем его выходной символический тензор (в коде это `truncatedBaseOutput`). Далее создаем новую верхушку модели, состоящую из одного плотного слоя. Форма входного сигнала новой верхушки соответствует форме символического тензора `truncatedBaseOutput`, а его выходная форма — числу слов в наборе данных для переноса обучения (пять в случае рис. 5.9). В параметрах этого плотного слоя указана многомерная логистическая функция активации, подходящая для задачи многоклассовой классификации. (Обратите внимание, что, в отличие от большинства остальных листингов кода в книге, следующий код написан на TypeScript. Если вы не знакомы с синтаксисом TypeScript, можете просто игнорировать такие нотации типов, как `void` и `tf.SymbolicTensor`.)

Листинг 5.7. Создание модели для переноса обучения в виде одного объекта tf.Model[1]

```
private createTransferModelFromBaseModel(): void {
  const layers = this.baseModel.layers;
  let layerIndex = layers.length - 2;
  while (layerIndex >= 0) {
    if (layers[layerIndex].getClassName().toLowerCase() === 'dense') {
      break;
    }
    layerIndex--;
  }
  if (layerIndex < 0) {
    throw new Error('Cannot find a hidden dense layer in the base model.');
  }
  this.secondLastBaseDenseLayer =
      layers[layerIndex];
  const truncatedBaseOutput = layers[layerIndex].output as
    tf.SymbolicTensor;

  this.transferHead = tf.layers.dense({
    units: this.words.length,
    activation: 'softmax',
    inputShape: truncatedBaseOutput.shape.slice(1)
  }));
  const transferOutput =
      this.transferHead.apply(truncatedBaseOutput) as tf.SymbolicTensor;
  this.model =
      tf.model({inputs: this.baseModel.inputs, outputs: transferOutput});
}
```

Находим предпоследний плотный слой базовой модели

Получаем слой, который разблокируем позднее, во время тонкой настройки (см. листинг 5.8)

Находим символический тензор

Создаем новую верхушку модели

«Применяем» новую верхушку к выходному сигналу усеченной базовой модели, для получения итогового выходного сигнала новой модели в виде символического тензора

С помощью API tf.model() создаем новую модель для переноса обучения, указывая в качестве ее входного сигнала входные сигналы исходной модели, а в качестве выходного сигнала — новый символический тензор

Мы по-новому используем новую верхушку: вызываем ее метод `apply()` с символическим тензором `truncatedBaseOutput` в качестве входного аргумента. Метод `apply()` доступен во всех объектах слоев и моделей в TensorFlow.js. Что же он делает? Как понятно из его названия, он «применяет» новую верхушку модели к входному сигналу и возвращает выходной сигнал. Важно понимать следующее.

- И входной и выходной сигналы представляют собой символические тензоры — заполнители для конкретных тензорных значений.
- На рис. 5.10 приведена наглядная иллюстрация этого: символический входной тензор (`truncatedBaseOutput`) не какая-то изолированная сущность, а выходной сигнал предпоследнего плотного слоя базовой модели. Этот плотный слой получает входные сигналы от другого слоя, который, в свою очередь, получает входные сигналы от расположенного ближе к началу слоя и т. д. Следовательно,

[1] Два примечания к этому листингу: 1) код написан на TypeScript, поскольку он входит в состав переиспользуемой библиотеки @tensorflow-models/speech-commands; 2) ради простоты мы удалили часть кода обработки ошибок.

`truncatedBaseOutput` несет в себе подграф базовой модели: а именно, подграф, заключенный между входным сигналом базовой модели и выходным сигналом предпоследнего плотного слоя. Другими словами, полный граф базовой модели, за исключением части, следующей за предпоследним плотным слоем. В результате выходной сигнал метода `apply()` несет граф, состоящий из этого подграфа, плюс новый плотный слой. В вызове функции `tf.model()`, возвращающей новую модель, используются совместно этот выходной сигнал и исходный входной сигнал. Новая модель совпадает с базовой, за исключением замены верхушки на новый плотный слой (см. рис. 5.10, *внизу*).

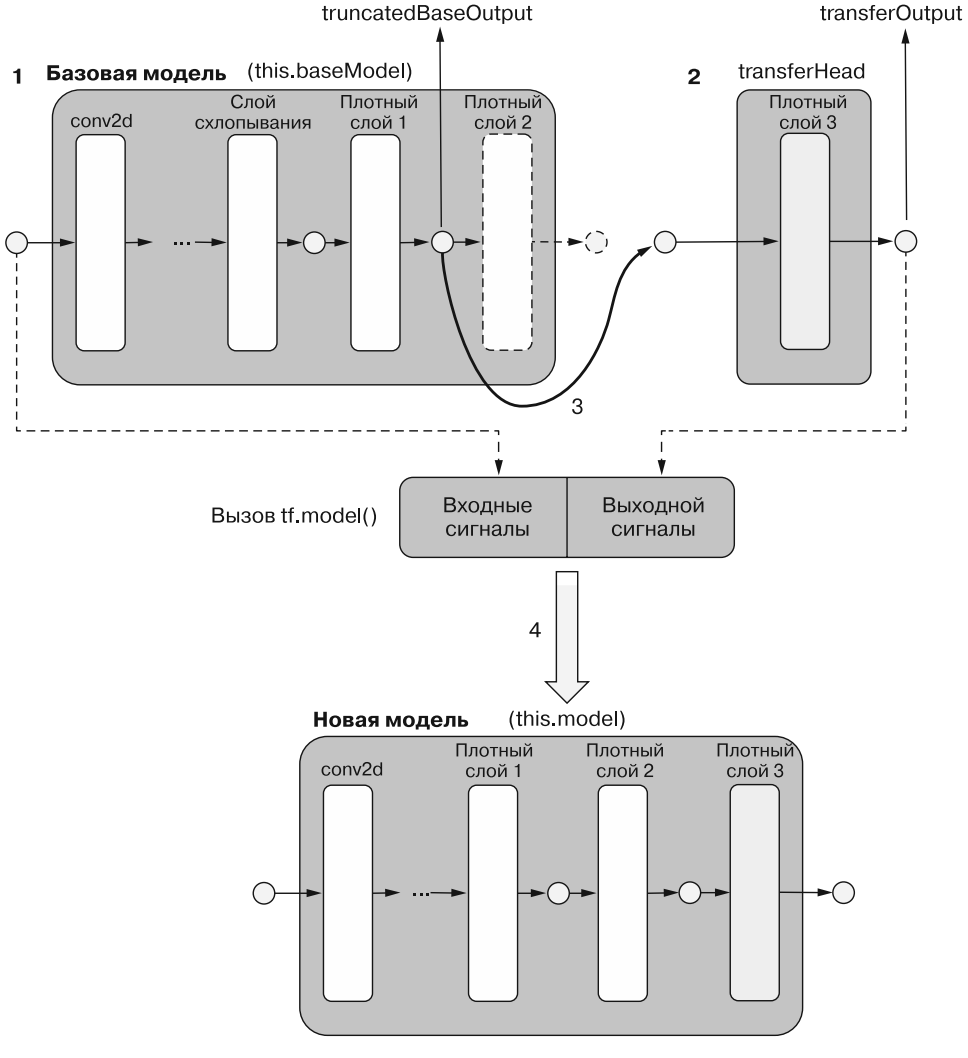


Рис. 5.10. Схематическая иллюстрация процесса создания новой сквозной модели для переноса обучения

Обратите внимание, что используемый здесь подход отличается от того, как мы объединяли модели в подразделе 5.1.2. Там мы создавали усеченную базовую модель и новую верхушку модели — два отдельных экземпляра. В результате выполнение вывода для каждого входного примера данных требовало двух вызовов `predict()`. Здесь же ожидаемые новой моделью входные сигналы идентичны тензорам аудиоспектрограмм, ожидаемым базовой моделью. В то же время новая модель выводит непосредственно оценки вероятностей для новых слов. Каждый вывод требует лишь одного вызова `predict()`, за счет чего ускоряется весь процесс. Благодаря инкапсуляции всех слоев в одной модели наш новый подход обладает еще одним преимуществом: возможностью обратного распространения ошибки через любые слои, участвующие в распознавании новых слов. Благодаря этому можно использовать прием тонкой настройки. Именно этим мы и займемся в следующем разделе.

Изучать рис. 5.10 следует вместе с листингом 5.7. Части данного рисунка, соответствующие переменным из листинга 5.7, обозначены моноширинным шрифтом. Шаг 1: получаем выходной символический тензор предпоследнего плотного слоя исходной модели (на него указывает широкая стрелка). Далее он будет использован на шаге 3. Шаг 2: создаем новую верхушку модели, состоящую из одного выходного плотного слоя (обозначенного на схеме как «Плотный слой 3»). Шаг 3: вызываем метод `apply()` новой верхушки модели, передавая в него символический тензор с шага 1 в качестве входного аргумента. Этот вызов связывает входной сигнал новой верхушки модели с усеченной базовой моделью с шага 1. Шаг 4: при вызове функции `tf.model()` совместно используются возвращаемое значение функции `apply()` и входной символический тензор исходной модели. Этот вызов возвращает новую модель, включающую все слои исходной модели от первого слоя до предпоследнего плотного слоя, в дополнение к плотному слою в новой верхушке. По существу, происходит замена старой верхушки исходной модели новой верхушкой, то есть закладывается фундамент для последующего обучения на данных, предназначенных для переноса. Обратите внимание, что часть (семь) слоев настоящей модели speech-command в схеме ради наглядности опущена. На этом рисунке закрашенные слои — обучаемые, а белые — нет.

Тонкая настройка путем разблокирования слоев

Тонкая настройка — необязательный шаг переноса обучения, следующий за начальным этапом обучения модели. На начальном этапе все слои базовой модели были заблокированы (их атрибуту `trainable` присвоено значение `false`), так что весовые коэффициенты обновлялись только у слоев верхушки. Мы уже видели такой тип начального обучения в примерах mnist-transfer-cnn и webcam-transfer-learning ранее в этой главе. Во время тонкой настройки разблокируется часть слоев базовой модели (их атрибут `trainable` равен `true`), после чего модель снова обучается на данных, предназначенных для переноса обучения. Это разблокирование слоев схематически показано на рис. 5.11. Реализация этого в TensorFlow.js для примера speech-command приведена в коде из листинга 5.8 (из файла `speech-commands/src/browser_fft_recognizer.ts`).

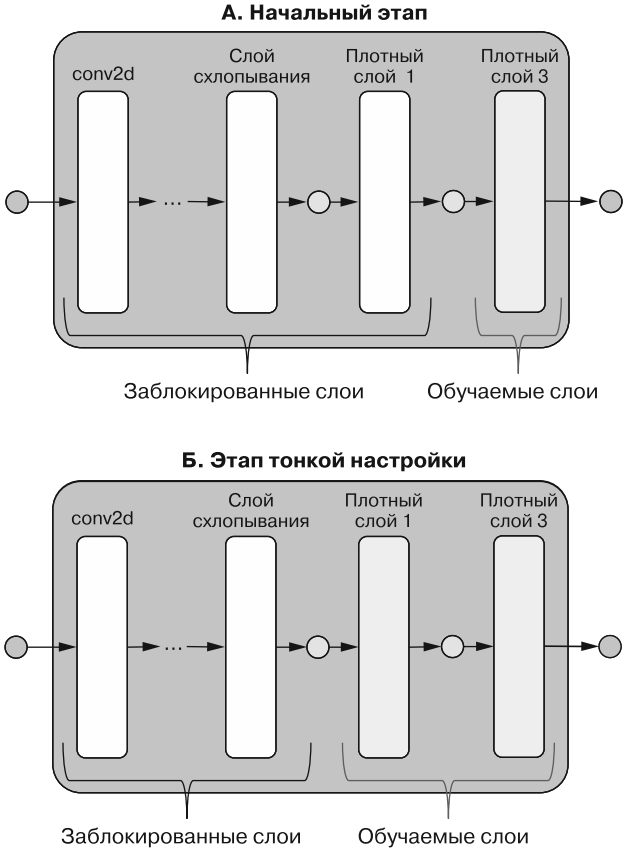


Рис. 5.11. Иллюстрируем заблокированные и разблокированные (то есть обучаемые) слои во время начального этапа переноса обучения (блок А) и этапа тонкой настройки (блок Б), (см. листинг 5.8). Обратите внимание, что плотный слой 3 следует непосредственно за плотным слоем 1 из-за усечения плотного слоя 2 (исходного выходного сигнала базовой модели), играющего роль первого шага переноса обучения (см. рис. 5.10)

Листинг 5.8. Начальный этап переноса обучения с последующей тонкой настройкой[1]

```
async train(config?: TransferLearnConfig):
      Promise<tf.History|[tf.History, tf.History]> {
    if (config == null) {
      config = {};
    }
    if (this.model == null) {
      this.createTransferModelFromBaseModel();
    }

    this.secondLastBaseDenseLayer.trainable = false;
```

Блокируем все слои усеченной базовой модели, включая тот, который мы позднее подвергнем тонкой настройке, на начальном этапе переноса обучения

[1] Мы удалили из листинга часть кода обработки ошибок, чтобы сосредоточиться на ключевых частях алгоритма.

```
this.model.compile({
  loss: 'categoricalCrossentropy',
  optimizer: config.optimizer || 'sgd',
  metrics: ['acc']
});

const {xs, ys} = this.collectTransferDataAsTensors();
let trainXs: tf.Tensor;
let trainYs: tf.Tensor;
let valData: [tf.Tensor, tf.Tensor];
try {
  if (config.validationSplit != null) {
    const splits = balancedTrainValSplit(
        xs, ys, config.validationSplit);
    trainXs = splits.trainXs;
    trainYs = splits.trainYs;
    valData = [splits.valXs, splits.valYs];
  } else {
    trainXs = xs;
    trainYs = ys;
  }

  const history = await this.model.fit(trainXs, trainYs, {
    epochs: config.epochs == null ? 20 : config.epochs,
    validationData: valData,
    batchSize: config.batchSize,
    callbacks: config.callback == null ? null : [config.callback]
  });

  if (config.fineTuningEpochs != null && config.fineTuningEpochs > 0) {
    this.secondLastBaseDenseLayer.trainable = true;

    const fineTuningOptimizer: string|tf.Optimizer =
        config.fineTuningOptimizer == null ? 'sgd' :
                                             config.fineTuningOptimizer;

    this.model.compile({
      loss: 'categoricalCrossentropy',
      optimizer: fineTuningOptimizer,
      metrics: ['acc']
    });

    const fineTuningHistory = await this.model.fit(trainXs, trainYs, {
      epochs: config.fineTuningEpochs,
      validationData: valData,
      batchSize: config.batchSize,
      callbacks: config.fineTuningCallback == null ?
          null :
          [config.fineTuningCallback]
    });
    return [history, fineTuningHistory];
```

Компилируем модель для начального этапа переноса обучения

При необходимости проверки разбиваем предназначенные для переноса обучения данные пропорционально на обучающий и проверочный наборы данных

Вызываем метод Model.fit() для начального переноса обучения

Для тонкой настройки разблокируем предпоследний плотный слой базовой модели (последний слой усеченной базовой модели)

Перекомпилируем модель после разблокирования вышеупомянутого слоя (иначе разблокирование ничего не даст)

Вызываем метод Model.fit() для тонкой настройки

```
      } else {
        return history;
      }
    } finally {
      tf.dispose([xs, ys, trainXs, trainYs, valData]);
    }
  }
```

Следует отметить несколько важных нюансов кода из листинга 5.8.

- После каждого блокирования или разблокирования слоев (изменяя значение их атрибута `trainable`) необходимо еще раз вызвать метод `compile()` модели, чтобы изменения вступили в силу. Мы уже упоминали это при обсуждении примера transfer-learning для MNIST в подразделе 5.1.1.
- Часть данных выделяется для проверки того, насколько хорошо полученные значения потерь и безошибочности отражают работу модели на данных, не виденных ею во время обратного распространения ошибки. Впрочем, мы выделяем часть собранных данных для проверки иначе, чем раньше, так что стоит остановиться на этом подробнее.
- В примере сверточной сети MNIST (см. листинг 4.2) последние 15–20 % данных выделялись на проверку с помощью параметра `validationSplit` метода `Model.fit()`. Здесь такой подход работает не слишком хорошо. Почему? А потому, что размер обучающего набора данных в таком случае намного меньше объемов данных в предыдущих примерах. В результате отделение вслепую нескольких последних примеров данных для проверки может привести к недостаточной представленности отдельных слов в проверочном подмножестве данных. Допустим, мы собрали по восемь примеров данных для каждого из четырех слов: *feel*, *seal*, *veal* и *zeal* — и взяли для проверки последние 25 % из 32 примеров данных (восемь примеров). В среднем на каждое слово в проверочном подмножестве приходится по два примера данных. Но из-за случайности выбора на какие-то слова в проверочном наборе придется лишь один пример данных, а на какие-то — вообще ни одного! Ясно, что не содержащий определенных слов проверочный набор данных плохо подходит для оценки безошибочности модели. Именно поэтому нам пришлось воспользоваться собственной функцией (`balancedTrainValSplit` в листинге 5.8), учитывающей истинные метки примеров данных и гарантирующей, что все слова будут представлены должным образом как в обучающем, так и в проверочном наборе. Рекомендуем вам поступать аналогичным образом для всех приложений с переносом обучения, в которых наборы данных столь же невелики.

Итак, что дает нам тонкая настройка? Какой дополнительный вклад она вносит после начального этапа переноса обучения? Для иллюстрации построим график кривых потерь и безошибочности с начального этапа и этапа тонкой настройки, объединив их в непрерывные кривые (рис. 5.12, блок A). Набор данных для переноса обучения здесь состоит из тех же четырех слов, которые мы уже видели на рис. 5.9. Первые 100 эпох каждой кривой соответствуют начальному этапу, а последние 300 — этапу тонкой настройки. Как видите, к концу 100 эпох начального

обучения кривые потерь и безошибочности начинают выравниваться и подчиняться закону убывающей доходности. Безошибочность на проверочном подмножестве данных выходит на постоянный уровень около 84 %. (Обратите внимание, насколько может ввести в заблуждение кривая безошибочности на одном только *обучающем* подмножестве данных, с легкостью приближающаяся к 100 %.) Впрочем, после разблокирования плотного слоя базовой модели, перекомпиляции модели и запуска этапа тонкой настройки безошибочность на проверочном наборе начинает расти и может достигать 90–92 % — весьма приличный 6–8%-ный прирост. Аналогичный эффект можно наблюдать на кривой потерь на проверочном наборе данных.

Для иллюстрации преимуществ тонкой настройки, по сравнению с переносом обучения без тонкой настройки, покажем на рис. 5.12, в блоке Б, что получится, если обучать перенесенную модель в течение тех же 400 эпох без тонкой настройки нескольких верхних слоев базовой модели. Не наблюдается никакой точки перегиба кривых потерь и безошибочности, как в блоке А на эпохе 100, когда начинает действовать тонкая настройка. Вместо этого кривые потерь и безошибочности выравниваются и сходятся к худшим значениям.

Чему же тонкая настройка обязана своим эффектом? Ее можно считать способом расширения разрешающих возможностей модели. За счет разблокирования части «верхних» слоев базовой модели перенесенная модель может минимизировать функцию потерь в пространстве параметров более высокой размерности, чем на начальном этапе. В чем-то это напоминает добавление в нейронную сеть скрытых слоев. Параметры разблокированного плотного слоя оптимизированы под исходный набор данных (состоящий из слов *one*, *two*, *yes* и *no*) и, возможно, плохо подходят для слов, участвующих в переносе. Дело в том, что внутренние представления, с помощью которых модель различала исходные слова, могут оказаться нерепрезентативными для распознавания слов в задаче переноса. Благодаря дальнейшей оптимизации (то есть тонкой настройке) этих параметров для слов из задачи переноса представление оптимизируется для работы со словами перенесенной модели. А значит, для этих слов резко возрастает безошибочность на проверочном наборе данных. Обратите внимание, что этот рост заметнее на более сложных задачах переноса обучения (как в случае четырех схожих по звучанию слов: *feel*, *seal*, *veal* и *zeal*). В более простых задачах (при четко различимых словах наподобие *red* и *green*) для достижения 100%-ной безошибочности на проверочном наборе данных нередко оказывается достаточно и начального этапа переноса обучения.

Возникает вопрос: в данном случае мы разблокировали только один слой базовой модели, и улучшатся ли результаты, если разблокировать еще несколько? Если кратко, то это зависит от многих нюансов, поскольку разблокирование дополнительных слоев еще больше расширяет разрешающие возможности модели. Но, как мы упоминали в главе 4 и обсудим подробнее в главе 8, чем больше разрешающие возможности модели, тем выше риск переобучения, особенно при небольшом наборе данных наподобие собранных нами в браузере примеров аудиоданных. И это не считая дополнительной вычислительной нагрузки, связанной с обучением дополнительных слоев. Можете поэкспериментировать с этим сами в упражнении 4 в конце главы.

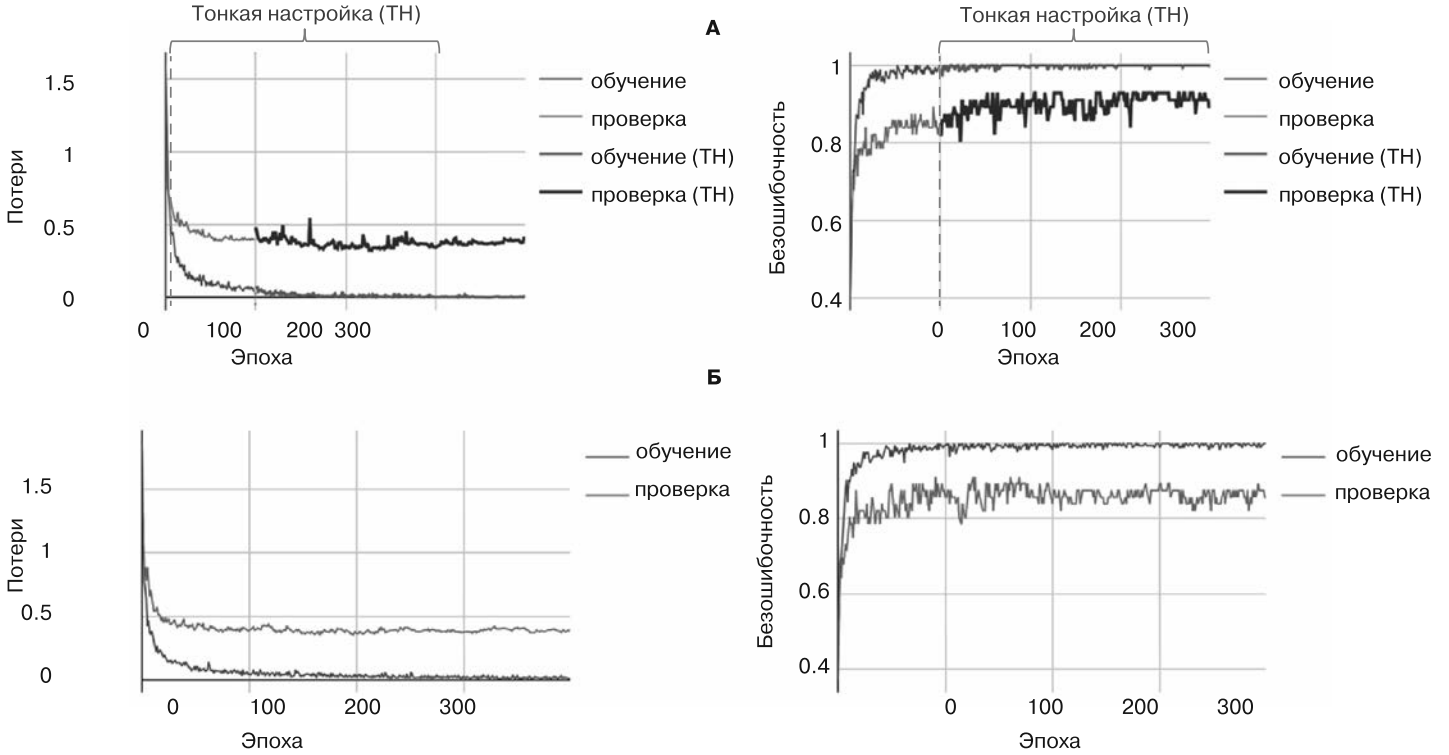


Рис. 5.12. Блок А: кривые потерь и безошибочности нашего примера при переносе обучения и последующей тонкой настройке (ТН в легендах графиков). Обратите внимание на точку перегиба на стыке частей кривых для начального этапа и тонкой настройки. Тонкая настройка ускоряет сокращение потерь и рост безошибочности благодаря разблокированию нескольких верхних слоев базовой модели, что ведет к росту разрешающих возможностей модели и ее приспособляемости к уникальным признакам в данных, предназначенных для переноса обучения. Блок Б: кривые потерь и безошибочности при обучении перенесенной модели в течение такого же числа эпох (400) без тонкой настройки. Обратите внимание, что без тонкой настройки функция потерь на проверочном наборе данных сходится к большему, а безошибочность на проверочном наборе — к меньшему значению, по сравнению с блоком А. Отметим также, что итоговый показатель безошибочности достигает примерно 0,9 при тонкой настройке (блок А), но застревает приблизительно на 0,85 при таком же числе эпох без тонкой настройки (блок Б)

Подытожим этот раздел, посвященный переносу обучения в TensorFlow.js. Вы познакомились с тремя различными способами переиспользования предобученной модели для новых задач. Для удобства выбора одного из них в ваших будущих проектах, связанных с переносом обучения, приведем в табл. 5.1 краткую сводку подходов, а также достоинств и недостатков каждого из них.

Таблица 5.1. Общая сводка трех подходов к переносу обучения в TensorFlow.js, их достоинств и недостатков

Подход	Достоинства	Недостатки
Использование исходной модели с блокированием ее первых нескольких (отвечающих за выделение признаков) слоев (см. подраздел 5.1.1)	Простой и удобный	Работает, только если форма выходного сигнала и функция активации, необходимые для перенесенной модели, такие же, как в базовой модели
Получение внутренних функций активации исходной модели в виде вложений для входного примера данных с созданием новой модели, принимающей вложение на входе (см. подраздел 5.1.2)	• Применим к сценариям переноса обучения, в которых требуется форма выходного сигнала, отличная от исходной. • Имеется прямой доступ к тензорам вложений, благодаря чему можно использовать, например, классификацию методом k-ближайших соседей (см. инфобокс 5.2)	• Необходимо работать с двумя отдельными экземплярами модели. • Есть сложности в тонкой настройке слоев исходной модели
Создание новой модели, включающей слои выделения признаков исходной модели и слои новой верхушки (см. подраздел 5.1.3)	• Применим к сценариям переноса обучения, в которых требуется форма выходного сигнала, отличная от исходной. • Имеется только один экземпляр модели. • Есть возможность тонкой настройки слоев выделения признаков	Отсутствует прямой доступ к внутренним функциям активации (вложениям)

5.2. Обнаружение объектов с помощью переноса обучения для сверточной сети

Представленные выше в главе примеры переноса обучения характеризуются одной общей чертой: суть задачи машинного обучения не меняется после переноса обучения. В частности, в них модель машинного зрения, обученная на задаче многоклассовой классификации, применялась к другой задаче многоклассовой классификации. В этом разделе мы покажем, что это не обязательно. Базовую модель можно использовать для решения совершенно иных, по сравнению с исходной, задач — например, обученную на задаче классификации базовую модель можно задействовать для выполнения регрессии (подбора числа). Подобный перенос обучения между разными

предметными областями — прекрасный пример гибкости и переиспользуемости глубокого обучения, главных причин широкого успеха этой сферы исследований.

Для иллюстрации воспользуемся новой задачей *обнаружения объектов* (object detection) — первым в книге типом задач машинного зрения, не связанных с классификацией. Обнаружение объектов связано с выявлением в изображении определенных классов объектов. В чем отличия от классификации? При обнаружении объектов результат представляет собой не только класс обнаруженного объекта (к какому типу он относится), но и дополнительную информацию относительно его расположения в изображении (где этот объект находится). Эту вторую половину информации простой классификатор не предоставляет. Например, в стандартной системе обнаружения объектов, используемой в беспилотных автомобилях, результаты анализа входного изображения системой включают не только типы присутствующих там объектов, которые нас интересуют (например, автомобилей и пешеходов), но и местоположение, видимый размер и пространственное расположение объектов в системе координат изображения.

Пример кода вы можете найти в каталоге `simple-object-detection` репозитория tfjs-examples. Учтите, что пример отличается от виденных вами ранее, поскольку обучение модели в Node.js в нем сочетается с выполнением вывода в браузере. Если точнее, обучение модели производится с помощью tfjs-node (или tfjs-node-gpu), после чего обученная модель сохраняется на диск. Далее для выдачи сохраненных файлов модели, наряду со статическими файлами `index.html` и `index.js`, используется сервер parcel — так мы продемонстрируем выполнение вывода на основе этой модели в браузере.

Для запуска примера используется такая последовательность команд (она включает несколько строк комментариев, которые при вводе можно опустить):

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/simple-object-detection
yarn
# Необязательный шаг для обучения собственной модели с помощью Node.js:
yarn train \
    --numExamples 20000 \
    --initialTransferEpochs 100 \
    --fineTuningEpochs 200
yarn watch  # Выполняем в браузере вывод для обнаружения объектов
```

Команда `yarn train` запускает обучение модели на локальной машине, по завершении которого сохраняет модель в каталоге `./dist`. Учтите, что эта операция занимает много времени и лучше выполнять ее с помощью GPU с поддержкой CUDA, ускоряющего обучение в 3–4 раза. Для этого необходимо лишь добавить в команду `yarn train` флаг `--gpu`:

```
yarn train --gpu \
    --numExamples 20000 \
    --initialTransferEpochs 100 \
    --fineTuningEpochs 200
```

Впрочем, если у вас нет времени или ресурсов, чтобы обучить модель на своей машине, не волнуйтесь: просто пропустите команду `yarn train` и перейдите непосредственно к `yarn watch`. На странице вывода, работающей в браузере, вы сможете загрузить уже обученную модель из централизованного хранилища по HTTP.

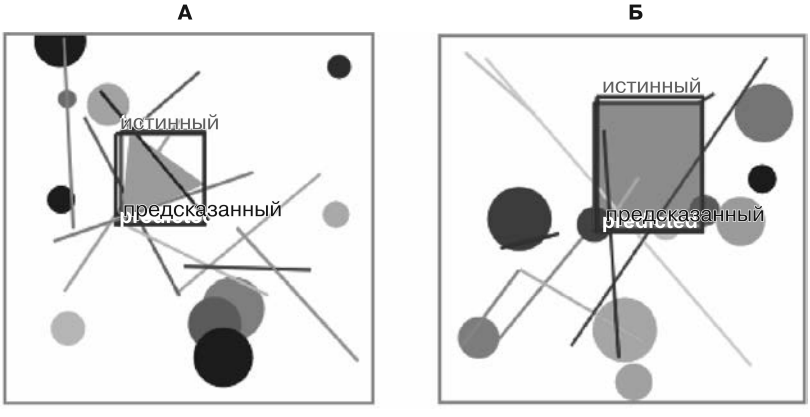
5.2.1. Задача обнаружения простых объектов в синтезированных изображениях

Современные методики обнаружения объектов включают множество приемов, неуместных в нашем начальном руководстве. Мы хотели бы показать сущность обнаружения объектов и не увязнуть при этом в болоте технических нюансов. Для этого мы создали задачу обнаружения простых объектов, включающую синтезированные изображения (рис. 5.13). Размерность этих синтезированных изображений — 224×224, а глубина представления цвета — 3 (RGB-каналы), то есть они совпадают со спецификациями входных данных модели MobileNet, лежащей в основе нашей модели. Как демонстрирует пример на рис. 5.13, у всех изображений белый фон. А роль объекта, который требуется обнаружить, играет либо равносторонний треугольник, либо прямоугольник. Если это треугольник, случайным образом варьируются размер и ориентация; если же объект — прямоугольник, случайным образом варьируются высота и ширина. Если бы изображение состояло лишь из белого фона и интересующего нас объекта, задача была бы слишком проста для демонстрации возможностей нашей методики. Чтобы усложнить ее, мы будем случайным образом вставлять в изображения определенное количество зашумляющих объектов — по десять кругов и десять отрезков в каждое. Местоположения и размеры кругов генерируются случайно, как и местоположения и длины отрезков. Часть этих зашумляющих объектов может накладываться на целевой объект, частично закрывая его. Цвета всех целевых и зашумляющих объектов выбираются случайным образом.

Полностью охарактеризовав входные данные, мы можем описать задачу для модели, которую создадим и будем обучать. Ее выходной сигнал — пять чисел, разбитых на две группы.

- Первая группа содержит одно число, указывающее, чем является обнаруженный объект — треугольником или прямоугольником (независимо от его расположения, размера, ориентации и цвета).
- Оставшиеся четыре числа составляют вторую группу и представляют собой координаты прямоугольника, ограничивающего обнаруженный объект. Если точнее, они представляют собой его координаты слева по оси *X*, справа по оси *X*, сверху по оси *Y* и снизу по оси *Y*. См. пример на рис. 5.13.

В искусственных данных хорошо то, что: 1) автоматически известны истинные метки и 2) можно сгенерировать столько данных, сколько нужно. При каждой генерации изображения тип объекта и его ограничивающий прямоугольник автоматически становятся нам известны из самого процесса генерации. Так что не нужно тратить усилия на маркирование обучающих изображений. Рекомендуем вам хорошо познакомиться с методикой этого весьма эффективного процесса, включающего совместный синтез входных признаков и меток и используемого во множестве сред тестирования и создания прототипов для моделей глубокого обучения. Однако обучение моделей обнаружения объектов для реальных входных изображений требует маркирования вручную реальных изображений. К счастью, уже существует множество подобных маркированных наборов данных. Один из них — набор данных COCO (Common Object in Context) (см. http://cocodataset.org).



Длительность вывода (мс): 36.7
Истинный класс объекта: **треугольник**
Предсказанный класс объекта: *треугольник*

Длительность вывода (мс): 26.0
Истинный класс объекта: **прямоугольник**
Предсказанный класс объекта: *прямоугольник*

Рис. 5.13. Пример синтезированных изображений для задачи обнаружения простых объектов. Блок А: роль целевого объекта играет повернутый равносторонний треугольник. Блок Б: роль целевого объекта играет прямоугольник. Прямоугольник с меткой true — истинный ограничивающий прямоугольник для интересующего нас объекта. Учтите, что интересующий нас объект иногда может оказаться частично закрыт некоторыми зашумляющими объектами (отрезками и кругами)

По завершении обучения модель сможет определить местоположение целевых объектов и классифицировать их с достаточно хорошей степенью безошибочности (как демонстрируют примеры на рис. 5.13). Чтобы лучше разобраться в том, как обучить модель решать эту задачу, заглянем в соответствующий код.

5.2.2. Углубляемся в обнаружение простых объектов

Создадим нейронную сеть для решения задачи обнаружения объектов в синтезированных изображениях. Как и ранее, в основу модели ляжет предобученная MobileNet ради использования широких возможностей выделения визуальных признаков ее сверточных слоев. Для этой цели в листинге 5.9 служит метод `loadTruncatedBase()`. Однако перед нашей моделью стоит новое испытание — два предсказания одновременно, а именно, определение формы целевого объекта и поиск его координат в изображении. Ранее мы не сталкивались с подобными задачами двойного предсказания. Для ее решения воспользуемся следующим приемом: будем выдавать на выходе модели тензор, инкапсулирующий оба этих предсказания, и придумаем новую функцию потерь для измерения того, насколько хорошо модель решает обе задачи одновременно. Мы *могли бы* обучить две отдельные модели: одну для классификации формы объекта, а вторую — для предсказания ограничивающего прямоугольника. Но две модели потребуют больше вычислительных ресурсов и памяти, чем одна, и не позволят нам использовать для обеих задач одни слои выделения признаков (следующий код можно найти в файле `simple-object-detection/train.js`).

Листинг 5.9. Описание модели для обнаружения простых объектов на основе усеченной MobileNet[1]

```
const topLayerGroupNames = [
    'conv_pw_9', 'conv_pw_10', 'conv_pw_11'];
const topLayerName =
    `${topLayerGroupNames[topLayerGroupNames.length - 1]}_relu`;

async function loadTruncatedBase() {
  const mobilenet = await tf.loadLayersModel(
      'https://storage.googleapis.com/' +
          'tfjs-models/tfjs/mobilenet_v1_0.25_224/model.json');

  const fineTuningLayers = [];
  const layer = mobilenet.getLayer(topLayerName);
  const truncatedBase =
      tf.model({
        inputs: mobilenet.inputs,
        outputs: layer.output
      });
  for (const layer of truncatedBase.layers) {
    layer.trainable = false;
    for (const groupName of topLayerGroupNames) {
      if (layer.name.indexOf(groupName) === 0) {
        fineTuningLayers.push(layer);
        break;
      }
    }
  }
  return {truncatedBase, fineTuningLayers};
}

function buildNewHead(inputShape) {
  const newHead = tf.sequential();
  newHead.add(tf.layers.flatten({inputShape}));
  newHead.add(tf.layers.dense({units: 200, activation: 'relu'}));
  newHead.add(tf.layers.dense({units: 5}));
  return newHead;
}

async function buildObjectDetectionModel() {
  const {truncatedBase, fineTuningLayers} = await loadTruncatedBase();

  const newHead = buildNewHead(truncatedBase.outputs[0].shape.slice(1));
  const newOutput = newHead.apply(truncatedBase.outputs[0]);
  const model = tf.model({
    inputs: truncatedBase.inputs,
    outputs: newOutput
  });

  return {model, fineTuningLayers};
}
```

Указываем, какие слои будут разблокироваться для тонкой настройки

Получаем ссылку на промежуточный слой: последний слой выделения признаков

Формируем усеченную MobileNet

Блокируем все слои выделения признаков для начального этапа переноса обучения

Отслеживаем слои, которые должны быть разблокированы во время тонкой настройки

Этот выходной сигнал длиной 5 состоит из индикатора формы объекта длиной 1 и ограничивающего прямоугольника длиной 4 (см. рис. 5.14)

Создаем новую верхушку модели для задачи обнаружения простых объектов

Формируем полную модель для обнаружения объектов путем сочетания новой верхушки модели с усеченной MobileNet

[1] Ради большей ясности мы удалили часть кода обработки ошибок.

Ключевая часть модели двойного предсказания в листинге 5.9 создана с помощью метода `buildNewHead()`. Схема работы модели приведена на рис. 5.14, *слева*. Новая верхушка состоит из трех слоев. Слой схлопывания меняет форму выходного сигнала у последнего сверточного слоя усеченной MobileNet так, чтобы можно было добавить к ней плотные слои. Первый плотный слой — скрытый, с нелинейностью типа ReLU. Второй плотный слой — итоговый выходной слой верхушки, а потому и итоговый выходной слой всей модели обнаружения объектов. Функция активации в этом слое — линейная по умолчанию. Данный слой является ключевым для понимания работы модели, поэтому рассмотрим его подробнее.

Как видно из кода, количество выходных нейронов итогового плотного слоя равно пяти. Что отражают эти пять чисел? Они объединяют предсказания формы целевого объекта и ограничивающего прямоугольника. Что интересно, их смысл определяет не сама модель, а используемая функция потерь. Ранее вы уже встречали различные виды функций потерь с понятными названиями наподобие `meanSquaredError`, подходящими для соответствующих задач машинного обучения (например, см. табл. 3.6). Впрочем, это лишь один из двух способов задания функций потерь в TensorFlow.js. Второй способ, который мы и будем здесь использовать, включает описание пользовательской JavaScript-функции, соответствующей определенной сигнатуре. Эта сигнатура имеет такой вид.

- Два входных аргумента: 1) истинные метки входных примеров данных и 2) соответствующие предсказания модели. Каждый из них представляет собой двумерный тензор, причем форма этих тензоров должна совпадать, а первое измерение отражать размеры батчей.
- Возвращаемое значение представляет собой скалярный тензор (тензор формы `[]`), значение которого представляет собой средние потери примеров данных батча.

Наша пользовательская функция потерь, написанная в соответствии с этой сигнатурой, приведена в листинге 5.10 и графически изображена в правой части рис. 5.14. Первый входной аргумент `customLossFunction` (`yTrue`) — тензор с истинными метками формы `[batchSize, 5]`. Первый входной аргумент (`yPred`) — выходное предсказание модели, имеет такую же форму, как и `yTrue`. Из пяти измерений по второй оси `yTrue` (пяти столбцов, если рассматривать его как матрицу) первое представляет собой индикатор 0-1 формы целевого объекта (0 означает треугольник, а 1 — прямоугольник), что определяется способом синтеза данных (см. `simple-object-detection/synthetic_images.js`). Оставшиеся четыре столбца описывают прямоугольник, ограничивающий целевой объект, точнее, значения левой, правой, верхней и нижней его координат, каждое из которых находится в диапазоне от 0 до `CANVAS_SIZE` (224). Число 224 соответствует высоте и ширине входных изображений, его источник — размер входного изображения модели MobileNet, лежащей в основе нашей модели.

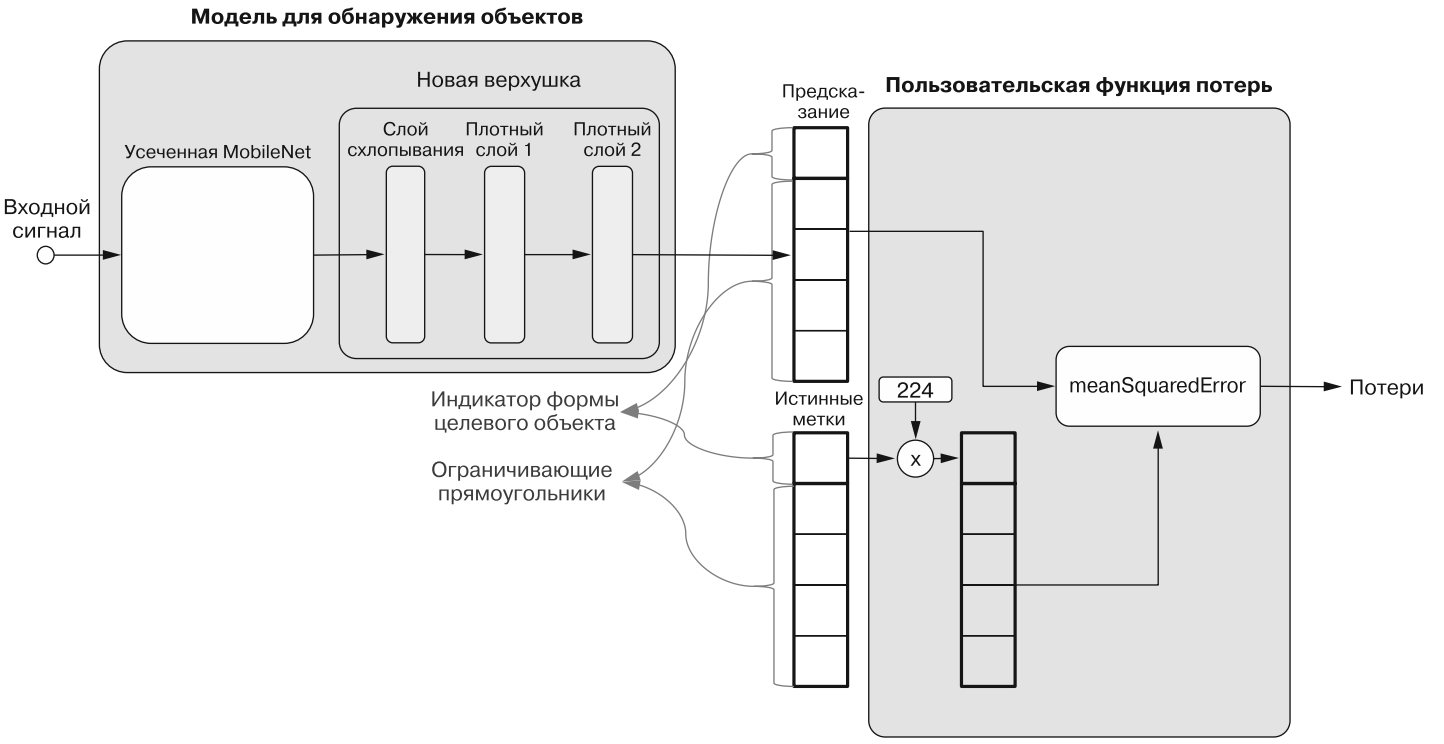


Рис. 5.14. Модель обнаружения объектов и лежащая в ее основе пользовательская функция потерь. Код формирования модели *(слева)* вы можете найти в листинге 5.9. Описание этой пользовательской функции потерь — в листинге 5.10

Листинг 5.10. Описание пользовательской функции потерь для задачи обнаружения объектов

```
const labelMultiplier = tf.tensor1d([CANVAS_SIZE, 1, 1, 1, 1]);
function customLossFunction(yTrue, yPred) {
  return tf.tidy(() => {
    return tf.metrics.meanSquaredError(
        yTrue.mul(labelMultiplier), yPred);
  });
}
```

Значения в столбце индикатора формы целевого объекта yTrue масштабируются на основе CANVAS_SIZE (224), чтобы гарантировать примерно одинаковый вклад в потери со стороны предсказания формы целевого объекта и предсказания ограничивающего прямоугольника

Наша пользовательская функция потерь получает на входе `yTrue` и масштабирует его первый столбец (индикатор 0-1 формы целевого объекта) в соответствии с `CANVAS_SIZE`, оставляя остальные столбцы неизменными. Далее она вычисляет среднеквадратичную погрешность (MSE) между значением `yPred` и масштабированным `yTrue`. Зачем мы масштабируем 0-1-метку формы целевого объекта в `yTrue`? Нам хотелось бы, чтобы выдаваемое моделью число отражало ее предсказание относительно формы целевого объекта — прямоугольник или треугольник. Если точнее, она выдает на выходе близкое к 0 число в случае треугольника и близкое к `CANVAS_SIZE` (224) число в случае прямоугольника. Так что во время вывода можно просто сравнить первое значение из выходного сигнала модели с `CANVAS_SIZE/2` (112), чтобы узнать, что предсказывает модель — форма объекта больше напоминает треугольник или прямоугольник. Вопрос в том, как оценить безошибочность этого предсказания формы целевого объекта и придумать функцию потерь. Наш ответ на этот вопрос: вычислить разницу между возвращаемым числом и индикатором 0-1, умноженным на `CANVAS_SIZE`.

Почему мы используем этот метод, а не бинарную перекрестную энтропию, как в примере с обнаружением фишинговых сайтов в главе 3? Дело в том, что здесь необходимо вычислять две метрики безошибочности: одну для предсказания формы целевого объекта, а вторую — для предсказания ограничивающего прямоугольника. Вторая задача требует предсказания непрерывных значений, ее можно считать задачей регрессии. Поэтому естественно будет выбрать для ограничивающих прямоугольников метрику MSE. Чтобы сочетать ее с другой метрикой, мы «притворяемся», что предсказание формы целевого объекта — тоже задача регрессии. Эта уловка позволяет нам использовать одну функцию метрики (вызов `tf.metric.meanSquaredError()` в листинге 5.10) для инкапсуляции функции потерь для обоих предсказаний.

Но зачем масштабировать индикатор 0-1 относительно `CANVAS_SIZE`? Если этого не сделать, модель в итоге будет генерировать числа в интервале 0–1 в качестве индикатора того, предсказывает ли модель, что форма целевого объекта — треугольник (ближе к 0) или прямоугольник (ближе к 1). Разница между числами в интервале `[0, 1]`, безусловно, намного меньше, чем разницы между координатами настоящего ограничивающего прямоугольника и предсказанных ограничивающих прямоугольников, расположенными в диапазоне от 0 до 224. В результате сигнал рассогласования от предсказания формы объекта будет совершенно незаметным по сравнению с сигналом рассогласования от предсказания ограничивающего прямоугольника,

что отнюдь не повысит безошибочность предсказания формы. Благодаря масштабированию индикатора 0-1 мы гарантируем равный вклад предсказаний формы объекта и ограничивающего прямоугольника в итоговое значение функции потерь (значение, возвращаемое функцией `customLossFunction()`), так что при обучении модели оба вида предсказаний будут оптимизироваться одновременно. В упражнении 4 в конце главы у вас будет возможность самим поэкспериментировать с этим масштабированием[1].

Когда данные подготовлены, а модель и функция потерь описаны, можно приступить к обучению модели! Основные части кода приведены в листинге 5.11 (из файла `simple-object-detection/train.js`). Как и показанная выше тонкая настройка (см. подраздел 5.1.3), обучение разбито на два этапа: начальный этап, в ходе которого слои новой верхушки обучаются вместе с несколькими верхними слоями усеченной базовой модели MobileNet, и этапом тонкой настройки, во время которого слои новой верхушки обучаются вместе с несколькими верхними слоями усеченной базовой модели MobileNet. Следует отметить, что непосредственно перед вызовом `fit()` для тонкой настройки необходимо вызвать (снова) метод `compile()`, чтобы вступили в силу изменения свойства `trainable` слоев. Если вы запустите обучение на своей машине, то легко заметите существенное падение значений функции потерь в начале этапа тонкой настройки. Оно отражает рост разрешающих возможностей модели и адаптацию разблокированных слоев выделения признаков к уникальным признакам в данных, предназначенных для обнаружения объектов, в результате их разблокирования. Список разблокируемых во время тонкой настройки слоев определяется массивом `fineTuningLayers`, заполняемым при усечении MobileNet (см. функцию `loadTruncatedBase()` в листинге 5.9), и содержит девять верхних слоев усеченной MobileNet. В упражнении 3 в конце главы вы сможете поэкспериментировать, разблокируя большее или меньшее число слоев базовой модели, и понаблюдать, как это влияет на безошибочность модели, получаемой в результате процесса обучения.

По завершении тонкой настройки модель сохраняется на диск, а затем загружается на этапе вывода, выполняемого в браузере (запускается командой `yarn watch`). Если вы загрузите модель, размещенную нами в Интернете, или потратите время и ресурсы на обучение достаточно хорошей модели на своей машине, то при выполнении вывода увидите неплохие предсказания (потери на проверочном наборе данных < 100 после 100 эпох начального обучения и 200 эпох тонкой настройки).

---

[1] Альтернатива масштабированию и основанному на meanSquaredError подходу — использовать первый столбец yPred в качестве показателя вероятности формы объекта и вычислять его бинарную перекрестную энтропию с первым столбцом yTrue. А затем суммировать это значение бинарной перекрестной энтропии с MSE, вычисленной по оставшимся столбцам yPred и yTrue. Но при таком альтернативном подходе необходимо масштабировать перекрестную энтропию должным образом, чтобы уравновесить ее с потерями для ограничивающего прямоугольника, как и при нашем текущем подходе. Для этого масштабирования необходим свободный параметр с тщательно выбранным значением. На практике он становится дополнительным гиперпараметром модели, который требует времени и вычислительных ресурсов для подбора — явный недостаток подхода. Ради простоты мы выбрали текущий подход.

Результаты вывода хороши, но не идеальны (см. примеры на рис. 5.13). Изучая их, учтите, что оценка, выполняемая в браузере, достаточно справедлива и отражает истинные возможности обобщения модели, поскольку примеры данных, подаваемые на вход этой обученной модели в браузере, отличаются от обучающих и проверочных примеров, виденных ею во время обучения.

Листинг 5.11. Второй этап обучения модели обнаружения объектов

```
const {model, fineTuningLayers} = await buildObjectDetectionModel();
model.compile({
  loss: customLossFunction,
  optimizer: tf.train.rmsprop(5e-3)
});

await model.fit(images, targets, {
  epochs: args.initialTransferEpochs,
  batchSize: args.batchSize,
  validationSplit: args.validationSplit
});

// Второй этап переноса обучения — тонкая настройка.

for (const layer of fineTuningLayers) {
  layer.trainable = true;
}
model.compile({
  loss: customLossFunction,
  optimizer: tf.train.rmsprop(2e-3)
});

await model.fit(images, targets, {
  epochs: args.fineTuningEpochs,
  batchSize: args.batchSize / 2,
  validationSplit: args.validationSplit
});
```

Для начального этапа задается относительно высокая скорость обучения

Выполнение начального этапа переноса обучения

Начало этапа тонкой настройки

Разблокирование части слоев для тонкой настройки

Задаем несколько более низкую скорость обучения для этапа тонкой настройки

На этапе тонкой настройки мы уменьшаем batchSize во избежание ситуаций нехватки памяти, вызванным повышенным, по сравнению с начальным этапом, количеством весовых коэффициентов, а значит, и потреблением памяти

Выполнение этапа тонкой настройки

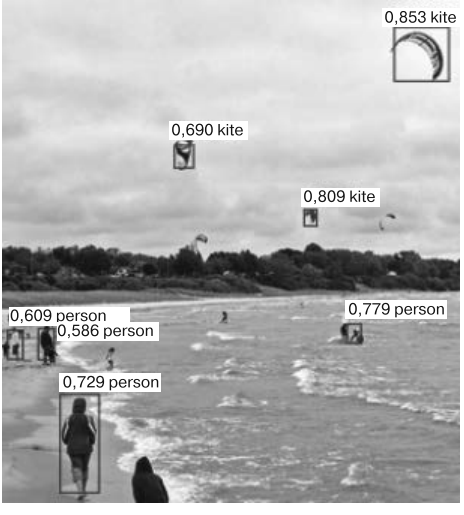
В завершение раздела мы показали, как успешно применить модель, обученную ранее для классификации изображений, к другой задаче: обнаружению объектов. При этом мы продемонстрировали, как описать пользовательскую функцию потерь, подходящую для двойственной сущности (классификации формы объекта + + регрессии для ограничивающего прямоугольника) задачи обнаружения объектов и как использовать ее во время обучения модели. Пример не только хорошо иллюстрирует основные принципы обнаружения объектов, но и подчеркивает гибкость переноса обучения, а также многообразие задач, для которых он применим. В реальных приложениях модели обнаружения объектов, конечно, намного сложнее, в них используется больше различных приемов, чем в нашем игрушечном примере с искусственно созданным набором данных. В инфобоксе 5.3 приводятся некоторые интересные факты о продвинутых моделях обнаружения объектов, их отличия от вышеприведенного простого примера, а также рассказывается, как их использовать с помощью TensorFlow.js.

ИНФОБОКС 5.3. Модели обнаружения объектов промышленного уровня

Обнаружение объектов — важная задача для многих типов приложений, включая приложения для интерпретации изображений, промышленной автоматизации и беспилотных автомобилей. В числе наиболее известных современных моделей обнаружения объектов — Single-Shot Detection[1] (SSD, пример для которой приведен на рисунке выше) и You Only Look Once (YOLO)[2]. Эти модели во многих аспектах схожи с моделью из нашего примера обнаружения простых объектов.

- Предсказывают как класс, так и местоположение объектов.
- Созданы на основе предобученных моделей классификации изображений наподобие MobileNet и VGG16[3] и обучены посредством переноса обучения.

Впрочем, во многих отношениях они и отличаются от нашей игрушечной модели.



Результат обнаружения объектов из TensorFlow.js-версии модели Single-Shot Detection (SSD). Обратите внимание на многочисленные ограничивающие прямоугольники и указанные для них классы объектов и степень уверенности модели

- Настоящие модели обнаружения объектов предсказывают намного больше классов объектов, чем наша простая модель (например, в наборе данных COCO — 80 категорий объектов; см. http://cocodataset.org/#home).
- Они способны обнаруживать несколько объектов в одном изображении (см. пример на рисунке выше).
- Архитектуры этих моделей намного сложнее, чем архитектура нашей простой модели. Например, модель SSD добавляет поверх усеченной предобученной модели несколько новых верхушек, чтобы предсказать степень уверенности модели в классах и ограничивающих прямоугольников для нескольких объектов во входном изображении.

---

1 *Liu W. et al.* SSD: Single Shot MultiBox Detector // Lecture Notes in Computer Science, 9905, 2016, http://mng.bz/G4qD.

2 *Redmon J. et al.* You Only Look Once: Unified, Real-Time Object Detection // Proceedings IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. Pp. 779–788. http://mng.bz/zlp1.

3 *Simonyan K., Zisserman A.* Very Deep Convolutional Networks for Large-Scale Image Recognition // submitted 4 Sept. 2014. https://arxiv.org/abs/1409.1556.

- В настоящих моделях обнаружения объектов в качестве функции потерь используется не отдельная метрика `meanSquaredError`, а взвешенная сумма двух типов функций потерь: 1) многомерной логистической функции перекрестной энтропии для показателей вероятности, предсказываемых для классов объектов и 2) функции потерь для ограничивающих прямоугольников наподобие `meanSquaredError` или `meanAbsoluteError`. Относительные веса для двух типов значений потерь тщательно подбираются для выравнивания вкладов со стороны обоих источников погрешности.
- Настоящие модели обнаружения объектов генерируют для каждого входного изображения множество кандидатов на роль ограничивающих прямоугольников. Далее лишние ограничивающие прямоугольники отсекаются, так что в итоговом выходном сигнале остаются только те, что обладают максимальными показателями вероятностей классов объектов.
- В некоторых настоящих моделях обнаружения объектов используются априорные знания о местоположении прямоугольников, ограничивающих объекты, — эмпирические предположения о возможном расположении ограничивающих прямоугольников в изображении, основанные на анализе большого числа маркированных реальных изображений. Благодаря этим априорным знаниям обучение моделей ускоряется, поскольку начинается не с произвольных случайных значений (как в примере simple-object-detection), а с некоего разумного начального состояния.

Лишь небольшое количество настоящих моделей обнаружения объектов было перенесено на TensorFlow.js. Например, поэкспериментировать с одной из лучших вы можете, заглянув в каталог `coco-ssd` репозитория tfjs-models. Чтобы увидеть ее в деле, выполните следующие команды:

```
git clone https://github.com/tensorflow/tfjs-models.git
cd tfjs-models/coco-ssd/demo
yarn && yarn watch
```

Чтобы узнать больше о настоящих, используемых на практике моделях обнаружения объектов, почитайте следующие посты блогов. Они касаются моделей SSD и YOLO соответственно, архитектуры и методики дополнительной обработки которых сильно различаются:

- Understanding SSD MultiBox — Real-Time Object Detection In Deep Learning (Eddie Forson): http://mng.bz/07dJ;
- Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3 (Jonathan Hui): http://mng.bz/KEqX.

До сих пор мы работали с уже готовыми для исследований наборами данных машинного обучения — отформатированными, очищенными благодаря кропотливой работе предыдущих исследователей и специалистов по машинному обучению. Очищенными до такой степени, что мы могли сосредоточить внимание на моделировании и не волноваться о вводе и предварительной обработке данных, равно как и об их корректности. Это справедливо для наборов данных MNIST и аудиоданных

в текущей главе и тем более для набора данных фишинговых сайтов и набора «Ирисы Фишера», которые мы использовали в главе 3.

Можно с уверенностью сказать, что *ни в одной* настоящей задаче машинного обучения не стоит рассчитывать на подобное везение. Львиную долю своего времени специалисты по машинному обучению тратят на добывание, предварительную обработку, очистку, проверку и форматирование данных[1]. В следующей главе мы расскажем, какие утилиты существуют в TensorFlow.js для упрощения процессов первичной обработки и ввода данных.

Упражнения

1. При обсуждении примера mnist-transfer-cnn в подразделе 5.1.1 мы упоминали, что задание свойства `trainable` слоев модели не сыграет никакой роли при обучении, если перед ним не вызвать метод `compile()` модели. Проверьте это утверждение, внеся определенные изменения в метод `retrainModel()` файла `index.js` данного примера.
   А. Вставьте перед строкой `this.model.compile()` вызов `this.model.summary()` и посмотрите на количество обучаемых и необучаемых параметров. Что они показывают? И чем отличаются от тех, которые получаются после вызова `compile()`?
   Б. Независимо от предыдущего пункта перенесите упомянутый вызов `this.model.compile()` прямо перед заданием свойства `trainable` слоев выделения признаков. Другими словами, установите указанное свойство этих слоев после вызова `compile()`. Как это повлияет на скорость обучения? Соответствует ли эта скорость случаю, когда обновляются лишь несколько последних слоев модели? Можете ли вы подтвердить другими способами, что в данном случае весовые коэффициенты первых нескольких слоев моделей обновляются во время обучения?
2. Во время переноса обучения в подразделе 5.1.1 (см. листинг 5.1) мы заблокировали первые два слоя conv2d, установив их свойство `trainable` равным `false` перед вызовом `fit()`. Добавьте в файл `index.js` примера mnist-transfer-cnn код для проверки того, что вызов `fit()` действительно не меняет весов слоев conv2d. Другой подход, с которым мы экспериментировали в том разделе, заключался в вызове `fit()` без блокирования слоев. Проверьте, на самом ли деле весовые коэффициенты слоев в данном случае меняются при вызове `fit()`. (Подсказка: вспомните, что в подразделе 2.4.2 для доступа к значениям весовых коэффициентов мы использовали атрибут `layers` объекта модели и его метод `getWeights()`.)
3. Преобразуйте приложение MobileNetV2 (не MobileNetV1! Мы уже это сделали) для Keras в формат TensorFlow.js и загрузите его в TensorFlow.js в браузере. Под-

[1] *Press G.* Cleaning Big Data: Most Time-Consuming, Least Enjoyable Data Science Task, Survey Says // Forbes, 23 Mar. 2016. http://mng.bz/9wqj.

робное описание шагов преобразования вы найдете в инфобоксе 5.1. Попробуйте воспользоваться методом `summary()`, чтобы посмотреть топологию MobileNetV2 и выяснить основные ее отличия от MobileNetV1.

4. Важный нюанс кода тонкой настройки в листинге 5.8 — повторный вызов метода `compile()` модели после разблокирования плотного слоя базовой модели. Выполните следующее.
   А. С помощью того же метода, что и в упражнении 2, убедитесь, что весовые коэффициенты (ядра и смещения) плотного слоя действительно не меняются при первом (на начальном этапе переноса обучения) вызове `fit()`, но меняются при втором (на этапе тонкой настройки).
   Б. Попробуйте закомментировать вызов `compile()`, следующий за строкой разблокирования (строкой кода, где меняется значение атрибута `trainable`), и посмотрите, как это повлияет на значения весов. Убедитесь, что вызов `compile()` действительно необходим, чтобы вступили в силу изменения состояний «заблокирован/разблокирован» слоев модели.
   В. Поменяйте код, попробовав разблокировать дополнительные слои базовой модели speech-command, включающие весовые коэффициенты (например, слой conv2d, предшествующий предпоследнему плотному слою), и посмотрите, как это повлияет на результаты тонкой настройки.
5. Для пользовательской функции потерь, описанной нами для задачи обнаружения простых объектов, мы масштабировали метку 0-1 формы объекта так, чтобы сигнал рассогласования от предсказания формы целевого объекта соответствовал по амплитуде сигналу рассогласования от предсказания ограничивающего прямоугольника (см. листинг 5.10). Посмотрите, что получится, если не выполнять такого масштабирования (убрав вызов `mul()` из кода в листинге 5.10). Убедитесь, что для достаточно точных предсказаний формы объекта такое масштабирование необходимо. Для этого можно также просто заменить экземпляры `customLossFunction` на `meanSquaredError` в вызове `compile()` (см. листинг 5.11). Учтите также, что при отказе от масштабирования во время обучения необходимо соответствующим образом изменить пороговое значение в ходе выполнения вывода, а именно поменять пороговое значение с `CANVAS_SIZE/2` на `1/2` в логике вывода (в файле `simple-object-detection/index.js`).
6. Этап тонкой настройки в примере обнаружения простых объектов включает разблокирование девяти верхних слоев усеченной базовой модели MobileNet (посмотрите, как заполняется массив `fineTuningLayers` в листинге 5.9). Естественно, возникает вопрос: почему именно девять? В этом упражнении поменяйте число разблокируемых слоев в массиве `fineTuningLayers` на большее/меньшее. Какие значения следующих величин вы ожидаете увидеть при разблокировании меньшего числа слоев при тонкой настройке: 1) итоговое значение функции потерь и 2) длительность каждой эпохи на этапе тонкой настройки? Насколько соответствуют ожидаемым результаты эксперимента? А в случае разблокирования большего числа слоев во время тонкой настройки?

Резюме

- Перенос обучения — процесс переиспользования предобученной модели или ее части для схожей задачи, но не идентичной той, для которой изначально обучалась данная модель. Подобное переиспользование ускоряет новое обучение.
- На практике при переносе обучения часто используются сверточные сети, обученные на очень больших наборах данных, предназначенных для классификации, наподобие сети MobileNet, обученной на наборе ImageNet. За счет одного размера исходного набора данных и разнообразия примеров данных в нем сверточные слои подобных предобученных моделей представляют собой мощные универсальные средства выделения признаков для широкого диапазона задач машинного зрения. Обучить подобные слои на небольшом количестве данных, доступном в типичных задачах переноса обучения, очень сложно, а то и вовсе невозможно.
- Мы обсудили несколько подходов к переносу обучения в TensorFlow.js, отличающихся друг от друга следующим: 1) создаются ли новые слои в качестве «новой верхушки» для переноса обучения и 2) производится ли перенос обучения с помощью одного экземпляра модели или двух. Каждый из этих подходов отличается своими достоинствами и недостатками и пригоден для различных сценариев использования (см. табл. 5.1).
- Путем задания атрибута `trainable` слоев модели можно предотвратить обновление их весовых коэффициентов во время обучения (вызовов `Model.fit()`). Эта методика, называемая блокированием, служит для «защиты» слоев выделения признаков в базовой модели во время переноса обучения.
- В некоторых задачах переноса обучения можно резко повысить скорость работы новой модели за счет разблокирования нескольких верхних слоев базовой модели после начального этапа обучения. Такое ускорение отражает адаптацию разблокированных слоев к уникальным признакам нового набора данных.
- Перенос обучения — универсальная и очень гибкая методика. Благодаря ему базовая модель может помочь при решении задач, отличных от той, для которой она изначально обучалась. Для иллюстрации этого мы продемонстрировали обучение модели обнаружения объектов, основанной на MobileNet.
- Функции потерь в TensorFlow.js можно описывать в виде пользовательских JavaScript-функций с тензорными входными и выходными значениями. Как мы показали в примере обнаружения простых объектов, на практике для решения задач машинного обучения часто оказываются необходимы пользовательские функции потерь.

Часть III

Продвинутые возможности глубокого обучения с TensorFlow.js

Если вы прочитали части I и II, то уже знакомы с основами глубокого обучения на TensorFlow.js. Часть III предназначена для тех пользователей, кто хотел бы лучше понять, что такое глубокое обучение, и овладеть его методиками. В главе 6 обсуждаются методики ввода и обработки, преобразования и работы с данными в контексте машинного обучения. В главе 7 описываются инструменты визуализации данных и моделей. В главе 8 мы сосредоточим внимание на важных проблемах недообучения и переобучения, а также способах их эффективного решения. Далее представим универсальный технологический процесс машинного обучения. В главах 9–11 обсуждаются практические вопросы трех продвинутых сфер глубокого обучения: последовательностей моделей, ориентированных на обработку, генеративных моделей и обучения с подкреплением соответственно. В этих главах вы познакомитесь с наиболее интересными функциями глубокого обучения.

6 Работа с данными

В этой главе

- Обучение моделей на больших наборах данных с помощью API `tf.data`.
- Исследование данных: поиск и исправление потенциальных проблем.
- Повышение качества модели за счет создания новых псевдопримеров данных с помощью дополнения данных.

Нынешней революцией машинного обучения мы во многом обязаны широкой доступности больших массивов данных. Без свободного доступа к большим объемам высококачественных данных такое взрывное развитие сферы машинного обучения было бы невозможно. Наборы данных сейчас доступны по всему Интернету, они свободно распространяются на таких сайтах, как Kaggle и OpenML, равно как и эталоны современных уровней производительности. Целые отрасли машинного обучения продвигаются вперед прежде всего за счет доступности «трудных» наборов данных, задавая планку и эталон для всего сообщества машинного обучения[1]. Если считать, что развитие машинного обучения — «космическая гонка» нашего времени, то данные можно считать «ракетным топливом»[2] благодаря их большому потенциалу, ценности, гибкости и критической важности для работы систем машинного обучения.

---

[1] Как, например, ImageNet привел к развитию сферы распознавания объектов или конкурс Netflix — сферы коллаборативной фильтрации.

[2] Эта аналогия взята из статьи: *Dumbill E.* Big Data Is Rocket Fuel // Big Data. Vol. 1. No. 2. Pp. 71–72.

Не говоря уже о том, что зашумленные данные, как и испорченное топливо, вполне могут привести к сбою системы. Вся эта глава посвящена данным. Мы рассмотрим рекомендуемые практики организации данных, обнаружения и исправления проблем в них, а также их эффективного использования.

«Но разве мы не работали с данными все это время?» — спросите вы. Да, в предыдущих главах мы работали с самыми разнообразными источниками данных. Мы обучали модели для изображений как на искусственных, так на и взятых с вебкамеры изображениях. Мы использовали перенос обучения для создания средства распознавания речи на основе набора аудиосемплов и брали данные из табличных наборов для предсказания цен. Что же здесь еще обсуждать? Разве мы не достигли мастерства в работе с данными?

Вспомните, какие паттерны использования данных встречались в предыдущих примерах. Обычно сначала нужно было скачать данные из удаленного источника. Далее мы (обычно) приводили их в нужный формат, например преобразовывали строки в унитарные векторы слов или нормализовали средние значения и дисперсии табличных источников данных. После этого мы организовывали данные в батчи и преобразовывали их в стандартные массивы чисел, представленные в виде тензоров, а затем уже подавали на вход модели. И это все еще до первого шага обучения.

Подобный паттерн скачивания — преобразования — организации по батчам очень распространен, и библиотека TensorFlow.js включает инструменты для его упрощения, модульной организации и снижения числа ошибок. В этой главе мы расскажем вам об инструментах из пространства имен `tf.data` и главном из них — `tf.data.Dataset`, позволяющем выполнять отложенную потоковую обработку данных. Благодаря этому подходу можно скачивать, преобразовывать данные и обращаться к ним по мере необходимости, вместо того чтобы скачивать источник данных полностью и хранить его в памяти для возможного доступа. Отложенная потоковая обработка существенно упрощает работу с источниками данных, не помещающимися в памяти отдельной вкладки браузера или даже в оперативной памяти машины.

Сначала мы познакомим вас с API `tf.data.Dataset` и покажем, как его настраивать и связывать с моделью. А затем приведем немного теории и расскажем об утилитах, предназначенных для просмотра и исследования данных с целью поиска и разрешения возможных проблем. Завершается глава рассказом о дополнении данных — методе расширения набора данных путем создания их искусственных псевдопримеров для повышения качества работы модели.

6.1. Работа с данными с помощью пространства имен tf.data

Как обучить спам-фильтр, если размер базы данных электронной почты занимает сотни гигабайт и база требует специальных учетных данных для доступа? Как создать классификатор изображений, если база данных обучающих изображений слишком велика и не помещается на одной машине?

Обращение к большим массивам данных и выполнение операций с ними — ключевой навык любого специалиста по машинному обучению, но до сих пор мы имели

дело лишь с приложениями, в которых данные прекрасно помещались в доступной приложению оперативной памяти. Множество приложений требуют работы с большими, громоздкими и, возможно, содержащими персональную информацию источниками данных, для которых подобная методика не подходит. Большие приложения требуют технологии доступа к данным, размещенным в удаленном источнике, по частям, по мере требования.

TensorFlow.js включает интегрированную библиотеку, предназначенную как раз для подобных операций с данными. Эта библиотека, вдохновленная API `tf.data` Python-версии TensorFlow, создана, чтобы пользователи могли с помощью коротких и удобочитаемых команд вводить данные, выполнять их предварительную обработку и переправлять их далее. Вся эта функциональность доступна в пространстве имен `tf.data`, если предварительно импортировать TensorFlow.js с помощью оператора следующего вида:

```
import * as tf from '@tensorflow/tfjs';
```

6.1.1. Объект tf.data.Dataset

Основная работа с модулем `tfjs-data` выполняется через единственный объект `tf.data.Dataset`. Он предоставляет простой, высокопроизводительный, с широкими возможностями настройки способ обхода и обработки больших (потенциально вообще неограниченных) списков элементов данных[1]. В самом первом приближении можно считать `Dataset` аналогом итерируемой коллекции произвольных элементов, в чем-то напоминающей `Stream` в Node.js. При запросе очередного элемента из `Dataset` внутренняя реализация скачивает его и обеспечивает доступ к нему либо при необходимости запускает функцию для его создания. Эта абстракция упрощает обучение модели на объемах данных, целиком не помещающихся в оперативной памяти, а также облегчает совместное использование и организацию объектов `Dataset` как полноправных объектов в тех случаях, когда их более одного. `Dataset` экономит память за счет потоковой передачи лишь требуемых битов данных, и не приходится обращаться ко всему массиву. API `Dataset` также оптимизирует работу, по сравнению с «наивной» реализацией, за счет упреждающей выборки значений, которые могут понадобиться.

6.1.2. Создание объекта tf.data.Dataset

По состоянию на версию 1.2.7 TensorFlow.js существует три способа подключения объекта `tf.data.Dataset` к поставщику данных. Мы довольно подробно рассмотрим их все, а краткую сводку вы найдете в табл. 6.1.

---

[1] В этой главе мы часто будем называть составляющие Dataset элементами (elements). В большинстве случаев термин «элемент» — синоним терминов «пример данных» (example) и «точка данных» (datapoint). То есть каждый элемент обучающего набора данных представляет собой пару (x, y). При чтении данных из CSV-источника элементы соответствуют строкам файла. Dataset достаточно гибок и позволяет работать с разнородными элементами, но делать это не рекомендуется.

Таблица 6.1. Создание объекта Dataset на основе источника данных

Способ получения нового объекта tf.data.Dataset	**API**	**Как с его помощью создать Dataset**
Из JavaScript-массива элементов; работает также для типизированных массивов, например Float32Array	`tf.data.array(items)`	`const dataset = tf.data.array([1,2,3,4,5]);` См. подробности в листинге 6.1

Из (возможно, удаленного) CSV-файла, в котором каждая строка соответствует элементу	`tf.data.csv(`
	`source,`
	`csvConfig)`
		`const dataset = tf.data.csv("https://path/to/my.csv");`
		См. подробности в листинге 6.2.
		Единственный обязательный параметр — URL источника данных. Кроме того, в необязательном параметре csvConfig можно передать объект с опциями для управления разбором CSV-файла. Например:
		• columnNames — если названия столбцов не указаны в заголовке или их нужно переопределить, можно передать их в объекте типа string[];
		• delimiter — строка из одного символа для переопределения разделителя по умолчанию (запятой);
		• columnConfigs — ассоциативный массив с ключами columnName для объектов columnConfig с целью управления разбором и возвращаемым типом Dataset. columnConfig сообщает средству синтаксического разбора тип элемента (строковое значение или целое число) либо указывает на необходимость рассматривать столбец как метку Dataset;
		• configuredColumnsOnly — указывает, возвращать ли данные для каждого столбца в CSV или только столбцов, определенных в объекте columnConfigs.
		Дополнительную информацию можно найти в документации API на сайте js.tensorflow.org

На основе обобщенной функции-генератора, выдающей элементы по одному	`tf.data.generator(`
	`generatorFunction)`
		`function* countDownFrom10() {`
		`for (let i=10; i>0; i--) {`
		`yield(i);`
		`}`
		`}`
		
		`const dataset =`
		`tf.data.generator(countDownFrom10);`
		См. подробности в листинге 6.3.
		Отметим, что tf.data.generator() при вызове без аргументов возвращает объект Generator

Создание объекта tf.data.Dataset из массива

Простейший способ создать новый объект `tf.data.Dataset` — сформировать его на основе JavaScript-массива. Создать `Dataset` на основе загруженного в память массива можно с помощью функции `tf.data.array()`. Конечно, не будет никакого выигрыша в скорости обучения или экономии памяти, по сравнению с непосредственным использованием массива, но у доступа к массиву через объект `Dataset` есть свои преимущества. Например, использование объектов `Dataset` упрощает организацию предварительной обработки, а также обучение и оценку благодаря простым API `model.fitDataset()` и `model.evaluateDataset()`, как мы увидим в разделе 6.2. В отличие от `model.fit(x, y)` вызов `model.fitDataset(myDataset)` не перемещает сразу все данные в память GPU, благодаря чему можно работать с наборами данных, не помещающимися туда целиком. Ограничение по памяти движка V8 JavaScript (1,4 Гбайт в 64-битных системах) обычно превышает объем, который TensorFlow.js может целиком разместить в памяти WebGL. Кроме того, использование API `tf.data` — одна из рекомендуемых практик инженерии разработки ПО, ведь оно упрощает модульный переход на другие типы данных без особых изменений кода. Без абстракции объекта `Dataset` подробности реализации источника данных могут легко просочиться в код его использования при обучении модели — узел, который придется распутывать при переходе на другую реализацию.

Для создания объекта `Dataset` из уже существующего массива можно воспользоваться `tf.data.array(itemsAsArray)`, как показано в листинге 6.1.

Листинг 6.1. Создание tf.data.Dataset из массива

```
const myArray = [{xs: [1, 0, 9], ys: 10},
                 {xs: [5, 1, 3], ys: 11},
                 {xs: [1, 1, 9], ys: 12}];
const myFirstDataset = tf.data.array(myArray);
await myFirstDataset.forEachAsync(
     e => console.log(e));

// Выдает примерно следующее
// {xs: Array(3), ys: 10}
// {xs: Array(3), ys: 11}
// {xs: Array(3), ys: 12}
```

Создает объект Dataset модуля tfjs-data, обеспечиваемый данными из массива. Учтите, что при этом не создается копий ни массива, ни его элементов

Обход всех значений объекта Dataset в цикле с помощью метода forEachAsync(). Учтите, что forEachAsync() — асинхронная функция, поэтому здесь необходимо ключевое слово await

Мы проходим по всем элементам `Dataset` в цикле с помощью функции `forEachAsync()`, по очереди выдающей все элементы. Больше подробностей о функции `forEachAsync()` вы можете найти в подразделе 6.1.3.

Элементы объектов `Dataset`, помимо тензоров, могут содержать простые типы данных JavaScript[1] (например, числа и строковые значения), а также кортежи,

[1] Если вы знакомы с реализацией модуля tf.data TensorFlow для Python, то, вероятно, удивитесь, что tf.data.Dataset, помимо тензоров, может содержать простые типы данных JavaScript.

массивы и многократно вложенные объекты подобных структур данных. В этом крошечном примере структура всех трех элементов объекта `Dataset` одинакова: это объекты с одинаковыми ключами и одним типом значений для ключей. В принципе, `tf.data.Dataset` позволяет комбинировать различные типы элементов, но наиболее распространен сценарий использования, при котором элементы `Dataset` представляют собой осмысленные семантические единицы одного типа. Обычно они отражают примеры одной сущности. Поэтому, за исключением очень необычных сценариев использования, типы данных и структуры всех элементов совпадают.

Создание объекта tf.data.Dataset из CSV-файла

Один из чаще всего встречающихся типов элементов в наборах данных — объект типа «ключ/значение», соответствующий одной строке таблицы, например одна строка CSV-файла. В листинге 6.2 приведена очень простая программа — она подключает и выводит набор данных Boston-housing, знакомый нам по главе 2.

Листинг 6.2. Создание объекта tf.data.Dataset из CSV-файла

```
  const myURL =
      "https://storage.googleapis.com/tfjs-examples/" +
          "multivariate-linear-regression/data/train-data.csv";
  const myCSVDataset = tf.data.csv(myURL);
  await myCSVDataset.forEachAsync(e => console.log(e));

// Выдает 333 строки вида
// {crim: 0.327, zn: 0, indus: 2.18, chas: 0, nox: 0.458, rm: 6.998,
// age: 45.8, tax: 222}
// ...
```

Создает объект Dataset модуля tfjs-data, обеспечиваемый данными из CSV-файла

Обход всех значений объекта Dataset в цикле с помощью метода forEachAsync(). Учтите, что forEachAsync() — асинхронная функция, поэтому здесь необходимо ключевое слово await

Здесь вместо `tf.data.array()` используется функция `tf.data.csv()`, в которую передается URL CSV-файла. В результате создается объект `Dataset`, обеспечиваемый данными из CSV-файла, проход в цикле по которому эквивалентен обходу в цикле строк CSV-файла. В Node.js можно подключиться к локальному CSV-файлу с помощью дескриптора URL с префиксом file://, вот так:

```
> const data = tf.data.csv(
    'file://./relative/fs/path/to/boston-housing-train.csv');
```

В цикле каждая строка CSV-файла преобразуется в объект JavaScript. Возвращаемые из объекта `Dataset` элементы представляют собой объекты, содержащие по одному свойству для каждого столбца CSV, причем эти свойства называются в соответствии с названиями столбцов CSV-файла, что удобно для работы с элементами, ведь теперь не требуется запоминать порядок полей. В подразделе 6.3.1 мы рассмотрим подробнее, как работать с CSV-файлами.

Создание объекта tf.data.Dataset на основе функции-генератора

Третий и наиболее гибкий способ создания `tf.data.Dataset` — на основе функции-генератора, для чего используется метод `tf.data.generator()`. Метод принимает в качестве аргумента *функцию-генератор* (`function*`[1]) языка JavaScript. Если вы не знакомы с функциями-генераторами — относительно новой возможностью JavaScript, то рекомендуем потратить немного времени на чтение документации. Цель функции-генератора — выдавать последовательность значений по мере необходимости, либо в бесконечном цикле, либо пока последовательность не закончится. Выдаваемые функцией-генератором значения превращаются в значения объекта `Dataset`. Например, простейшая функция-генератор может выдавать случайные числа или извлекать «снимки» состояния подключенного аппаратного устройства. Сложные функции-генераторы могут интегрироваться в компьютерные игры, выдавая снимки экрана, игровой счет, а также управлять вводом/выводом. В листинге 6.3 очень простая функция-генератор выдает элементы выборки бросания костей.

Листинг 6.3. Создание объекта tf.data.Dataset для случайных бросков костей

```
let numPlaysSoFar = 0;

function rollTwoDice() {
  numPlaysSoFar++;
  return [Math.ceil(Math.random() * 6), Math.ceil(Math.random() * 6)];
}

function* rollTwoDiceGeneratorFn() {
  while(true) {
    yield rollTwoDice();
  }
}

const myGeneratorDataset = tf.data.generator(
    rollTwoDiceGeneratorFn);
await myGeneratorDataset.take(1).forEachAsync(
    e => console.log(e));

// Выводит в консоль значение наподобие
// [4, 2]
```

Значение numPlaysSoFar наращивается в rollTwoDice(), благодаря чему мы можем подсчитать, сколько раз эта функция вызывалась объектом Dataset

Описываем функцию-генератор (с помощью синтаксиса function*), выдающей результат вызова rollTwoDice() неограниченное число раз

Здесь создается объект Dataset

Выборка ровно одного элемента объекта Dataset. Метод take(1) мы опишем в подразделе 6.1.4

Несколько любопытных замечаний относительно объекта `Dataset` для имитации игры из листинга 6.3. Во-первых, обратите внимание, что созданный здесь набор данных, `myGeneratorDataset`, бесконечен. Поскольку в функции-генераторе отсутствует `return`, можно спокойно производить выборку элементов из набора данных до бесконечности. Выполнение для этого набора данных `forEachAsync()` или `toArray()` (см. подраздел 6.1.3) никогда бы не завершилось и привело бы, по всей видимости,

[1] Узнать больше о функциях-генераторах ECMAscript вы можете по адресу http://mng.bz/Q0rj.

к аварийному сбою сервера или браузера, так что будьте осторожны! Для работы с подобными объектами необходимо создать другой объект `Dataset` — ограниченную выборку из неограниченного первоисточника, для чего следует воспользоваться `take(n)`. Чуть позже мы расскажем об этом подробнее.

Во-вторых, учтите, что объект `Dataset` производит замыкание локальной переменной. Это помогает при журналировании и отладке, позволяя определить число произведенных вызовов функции-генератора.

В-третьих, учтите, что данные до момента их запроса не существуют. В нашем случае мы за все время обращаемся только к одному элементу набора данных, что и отразится в значении `numPlaysSoFar`.

Наборы данных на основе генераторов отличаются широкими возможностями и исключительной гибкостью, позволяя разработчикам подключать модели к разнообразным API поставщиков данных, например получать данные из запроса к БД, из скачиваемых частями по сети данных или от подключенного аппаратного обеспечения. Подробнее API `tf.data.generator()` рассматривается в инфобоксе 6.1.

ИНФОБОКС 6.1. Спецификация аргументов функции tf.data.generator()

API `tf.data.generator()` — очень гибкий, обладает большими возможностями. С его помощью пользователи могут подключать модели к разнообразным поставщикам данных. Передаваемый в `tf.data.generator()` аргумент должен удовлетворять следующим спецификациям.

- Должен вызываться без аргументов.
- При вызове без аргументов он должен возвращать объект, удовлетворяющий протоколу итератора и итерируемого объекта. Это значит, что у этого возвращаемого объекта обязан быть метод `next()`. При вызове без аргументов метод `next()` должен возвращать JavaScript-объект `{value: ELEMENT, done: false}` для передачи далее значения `ELEMENT`. Когда больше нечего возвращать, он должен вернуть `{value: undefined, done: true}`.

Функции-генераторы JavaScript возвращают объекты типа `Generator`, удовлетворяющие этой спецификации, а значит, они предоставляют самый удобный способ использования API `tf.data.generator()`. Такая функция может служить замыканием для локальных переменных, обращаться к локальным аппаратным устройствам, подключаться к сетевым ресурсам и т. д.

В табл. 6.1 приводился следующий код, иллюстрирующий использование API `tf.data.generator()`:

```
function* countDownFrom10() {
  for (let i=10; i>0; i--) {
    yield(i);
  }
}

const dataset =
tf.data.generator(countDownFrom10);
```

Если вы по каким-либо причинам не хотите использовать функции-генераторы и предпочитаете вместо этого напрямую реализовать протокол итерируемого объекта, можете написать вышеприведенный код следующим эквивалентным образом:

```
function countDownFrom10Func() {
  let i = 10;
  return {
    next: () => {
      if (i > 0) {
        return {value: i--, done: false};
      } else {
        return {done: true};
      }
    }
  }
}

const dataset = tf.data.generator(countDownFrom10Func);
```

6.1.3. Доступ к данным в объекте Dataset

При наличии данных в объекте `Dataset`, разумеется, хочется каким-либо образом получить к ним доступ. Структуры данных, которые можно создать, но из которых нельзя ничего прочитать, не слишком полезны. Существует два API для извлечения данных из объекта `Dataset`, но пользователям `tf.data` редко приходится их применять. Обычно за доступ к данным в объекте `Dataset` отвечают более высокоуровневые API. Например, при обучении модели мы используем описанный в разделе 6.2 API `model.fitDataset()`. Он обращается к данным в объекте `Dataset` вместо нас, а нам как пользователям никогда не приходится обращаться к данным напрямую. Тем не менее для отладки, тестирования и анализа работы объекта `Dataset` важно понимать, что у него внутри.

Первый способ доступа к данным в объекте `Dataset` — их потоковый вывод в массив с помощью функции `Dataset.toArray()`, которая делает именно то, что и подразумевает ее название: проходит в цикле по всему объекту `Dataset`, перемещая все элементы в массив, и возвращает этот массив пользователю. Пользователям следует с осторожностью выполнять эту функцию, чтобы случайно не создать массив, слишком большой для среды выполнения JavaScript. Это распространенная ошибка, в случае, например, когда объект `Dataset` подключен к большому удаленному источнику данных или представляет собой неограниченный `Dataset`, предназначенный для чтения данных с датчика.

Второй способ доступа к данным в `Dataset` — выполнение некой функции для каждого примера данных этого `Dataset` с помощью `dataset.forEachAsync(f)`. Передаваемый `forEachAsync(f)` аргумент `f` применяется ко всем элементам по очереди аналогично конструкции `forEach()` в массивах и множествах JavaScript, то есть нативным `Array.forEach()` и `Set.forEach()`.

Важно отметить, что и `Dataset.forEachAsync()`, и `Dataset.toArray()` — асинхронные функции, в отличие от синхронной `Array.forEach()`, и допустить ошибку здесь очень легко. `Dataset.toArray()` возвращает промис и в общем случае требует ключевого слова `await` либо `.then()`, если от него ожидается синхронное поведение. Учтите, что, если забыть `await`, промис может не разрешиться так, как нужно, что приведет к ошибкам в работе программы. Одна из типичных ошибок связана с тем, что объект `Dataset` кажется пустым, поскольку обход его содержимого происходит до разрешения промиса.

Функция `Dataset.forEachAsync()` асинхронная, в отличие от синхронной `Array.forEach()`, вовсе не потому, что данные, к которым обращается `Dataset`, приходится создавать, вычислять или получать из удаленного источника. Асинхронность в этом случае дает возможность эффективно использовать доступные вычислительные ресурсы во время ожидания. Общая сводка этих методов приведена в табл. 6.2.

Таблица 6.2. Методы обхода объекта Dataset в цикле

Метод экземпляра объекта tf.data.Dataset	Что делает	Пример

`.toArray()`	Асинхронно проходит в цикле по всему объекту Dataset, перемещая все элементы в возвращаемый затем массив	`const a = tf.data.array([1, 2, 3, 4, 5, 6]);`
		`const arr = await a.toArray();`
		`console.log(arr);`
		
		`// 1,2,3,4,5,6`

`.forEachAsync(f)`	Асинхронно проходит в цикле по всем элементам объекта Dataset, применяя к каждому функцию f	`const a = tf.data.array([1, 2, 3]);`
		`await a.forEachAsync(e => console.log("hi " + e));`
		
		`// hi 1`
		`// hi 2`
		`// hi 3`

6.1.4. Операции над наборами данных модуля tfjs-data

Безусловно, очень удобно, если данные можно использовать в исходном виде, без какой-либо очистки или предварительной обработки. Но, по нашему личному опыту, такого *практически никогда* не случается, за исключением примеров, специально создаваемых в учебных целях или для оценки производительности. Чаще всего данные приходится каким-либо образом преобразовывать перед их анализом или применением в задачах машинного обучения. Например, источники данных нередко содержат лишние элементы, которые нужно отфильтровать. Зачастую данные, относящиеся к некоторым ключам, требуют разбора, десериализации или

переименования. Данные могут храниться в отсортированном виде, а значит, их нужно перетасовать случайным образом, прежде чем использовать для обучения или оценки качества модели. Набор данных может требовать разбиения на непересекающиеся множества для обучения и контроля. Предварительная обработка практически неизбежна. Если вам попался чистый и готовый к использованию набор данных — скорее всего, кто-то уже очистил и предварительно обработал их вместо вас!

`tf.data.Dataset` предоставляет предназначенные для подобных операций методы (табл. 6.3), которые можно организовывать цепочкой. Все они возвращают новый объект `Dataset`, но не думайте, что все элементы набора данных копируются или при каждом вызове метода все элементы обходятся в цикле! API `tf.data.Dataset` просто загружает и преобразует элементы отложенным образом. Объект `Dataset`, созданный соединением цепочкой нескольких из этих методов, можно считать маленькой программой, выполняемой только при запросе элементов на конце цепочки. Только в этот момент экземпляр `Dataset` проходит обратно по цепочке операций, возможно, прямо до запроса данных из удаленного источника данных.

Таблица 6.3. Методы объекта tf.data.Dataset, допускающие организацию цепочкой

Метод экземпляра объекта tf.data.Dataset	Что делает	Пример
`.filter(predicate)`	Возвращает объект Dataset, включающий только те элементы, для которых результат вычисления предиката равен true	`myDataset.filter(x => x < 10);` Возвращает объект Dataset, содержащий лишь те значения из myDataset, которые меньше 10
`.map(transform)`	Применяет указанную функцию к каждому из элементов объекта Dataset и возвращает полученный в результате новый объект Dataset	`myDataset.map(x => x * x);` Возвращает объект Dataset, состоящий из квадратов значений исходного объекта Dataset
`.mapAsync(` `asyncTransform)`	Аналогичен методу map, но передаваемая в него функция должна быть асинхронной	`myDataset.mapAsync(fetchAsync);` Если fetchAsync — асинхронная функция, выдающая извлеченные по указанному URL значения, то этот вызов вернет новый объект Dataset, содержащий соответствующие значения

`.batch(`
`batchSize,`
`smallLastBatch?)`
	Объединяет последовательные диапазоны элементов в цельные группы и преобразует элементы простых типов данных в тензоры	`const a = tf.data.array(`
		`[1, 2, 3, 4, 5, 6, 7, 8])`
		`.batch(4);`
		`await a.forEach(e => e.print());`
		
		`// Выводит:`
		`//  Tensor [1, 2, 3, 4]`
		`//  Tensor [5, 6, 7, 8]`

Метод экземпляра объекта tf.data.Dataset	Что делает	Пример
`.concatenate(dataset)`	Склеивает элементы двух объектов Dataset вместе в новый Dataset	`myDataset1.concatenate(myDataset2)` Возвращает объект Dataset, который обходит в цикле сначала все значения объекта myDataset1, а затем все значения объекта myDataset2
`.repeat(count)`	Возвращает объект Dataset, предназначенный для многократного (возможно, неограниченное число раз) обхода исходного объекта Dataset	`myDataset.repeat(NUM_EPOCHS)` Возвращает объект Dataset, который обходит в цикле по всем значениям объекта myDataset NUM_EPOCHS раз. Если параметр NUM_EPOCHS не указан или меньше нуля, выполняется неограниченное число обходов
`.take(count)`	Возвращает объект Dataset, содержащий только первые count примеров данных исходного	`myDataset.take(10);` Возвращает объект Dataset, содержащий только первые десять элементов объекта myDataset. Если myDataset содержит менее десяти элементов, то возвращается точно такой же объект Dataset
`.skip(count)`	Возвращает объект Dataset, из которого исключены первые count примеров данных исходного	`myDataset.skip(10);` Возвращает объект Dataset, содержащий все элементы myDataset, за исключением первых десяти. Если myDataset содержит десять элементов или меньше, возвращается пустой объект Dataset

`.shuffle(bufferSize, seed?)`	Генерирует объект Dataset, перетасовывающий элементы исходного объекта Dataset.
	Учтите: элементы при этой перетасовке выбираются случайным образом внутри окна размера bufferSize, так что упорядоченность вне этого окна не меняется
		`const a = tf.data.array([1, 2, 3, 4, 5, 6]).shuffle(3);`
		`await a.forEach(e => console.log(e));`
		`// Может вывести, например:`
		`// 2, 4, 1, 3, 6, 5`
		Выводит значения от 1 до 6, перетасованные случайным образом. Перетасовка частичная в том смысле, что возможна не любая упорядоченность, поскольку окно меньше общего размера данных. Например, последний элемент, 6, не может быть первым при новой упорядоченности, поскольку его пришлось бы сдвинуть назад на более чем bufferSize (3) позиций

Эти операции можно связывать цепочкой, создавая простые, но обладающие широкими возможностями конвейеры обработки. Например, для разбиения случайным образом набора данных на обучающий и контрольный наборы можно воспользоваться рецептом из листинга 6.4 (см. `tfjs-examples/iris-fitDataset/data.js`).

Листинг 6.4. Разбиение на обучающий/контрольный наборы данных с помощью tf.data.Dataset

```
const seed = Math.floor(
    Math.random() * 10000);
const trainData = tf.data.array(IRIS_RAW_DATA)
     .shuffle(IRIS_RAW_DATA.length, seed);
     .take(N);
     .map(preprocessFn);
const testData = tf.data.array(IRIS_RAW_DATA)
     .shuffle(IRIS_RAW_DATA.length, seed);
     .skip(N);
     .map(preprocessFn);
```

Используем одно и то же начальное значение перетасовки для обучающих и контрольных данных; иначе они перетасовывались бы независимо друг от друга и некоторые примеры данных могли бы попасть и туда, и туда

Пропускаем первые N примеров данных для получения контрольных данных

Берем первые N примеров в качестве обучающих данных

В этом листинге важно обратить внимание на следующее. Чтобы распределить примеры данных случайным образом по обучающему и контрольному наборам, мы сначала перетасовываем данные. Первые N примеров берем в качестве обучающих данных. А для получения контрольных данных пропускаем эти N примеров и берем остальные. Очень важно перетасовывать данные *одинаково* при выборке, чтобы один и тот же пример данных не оказался в обоих множествах; поэтому при выборке из обоих конвейеров используется одинаковое начальное значение для перетасовки.

Важно также отметить, что функция `map()` применяется *после* операции `skip`. Вызвать `.map(preprocessFn)` можно и до `skip`, но при этом `preprocessFn` будет выполняться и для отброшенных примеров данных — пустая трата вычислительных ресурсов. Проверить, что все происходит именно так, можно с помощью кода из листинга 6.5.

Листинг 6.5. Иллюстрация взаимодействия skip() и map() для Dataset.forEach

```
let count = 0;

// Тождественная функция, наращивающая также счетчик count
function identityFn(x) {
  count += 1;
  return x;
}

console.log('skip before map');
await tf.data.array([1, 2, 3, 4, 5, 6])
   .skip(6)
  .map(identityFn)
  .forEachAsync(x => undefined);
console.log(`count is ${count}`);

console.log('map before skip');
await tf.data.array([1, 2, 3, 4, 5, 6])
   .map(identityFn)
  .skip(6)
  .forEachAsync(x => undefined);
console.log(`count is ${count}`);
```

Сначала пропускаем значения, затем отображаем оставшиеся

Сначала отображаем значения, затем пропускаем

```
// Выводит:
// skip before map
// count is 0
// map before skip
// count is 6
```

Еще один распространенный сценарий использования `dataset.map()` — нормализация входных данных. Например, легко представить себе сценарий, в котором может пригодиться нормализация данных до нулевого среднего значения, но число входных примеров данных бесконечно. Для вычитания среднего значения необходимо сначала вычислить математическое ожидание распределения, но как вычислить среднее значение бесконечного множества? Можно было бы рассчитать среднее значение репрезентативной выборки из этого распределения, но, если взять выборку неправильного размера, легко допустить ошибку. Например, представьте себе распределение, почти все значения которого равны 0 и лишь значение каждого десятимиллионного примера данных равно 10^9. Математическое ожидание такого распределения равно 100, но, если вычислить среднее значение первого миллиона примеров данных, результат получится совершенно неправильный.

Можно выполнить потоковую нормализацию с помощью API `Dataset` следующим образом (листинг 6.6). В листинге подсчитывается скользящий итог числа просмотренных примеров данных, а также их скользящая сумма. Благодаря этому возможна потоковая нормализация. Здесь мы работаем со скалярными значениями, не тензорами, но структура версии для тензоров будет выглядеть аналогично.

Листинг 6.6. Потоковая нормализация с помощью метода tf.data.map()

```
function newStreamingZeroMeanFn() {
  let samplesSoFar = 0;
  let sumSoFar = 0;

  return (x) => {
    samplesSoFar += 1;
    sumSoFar += x;
    const estimatedMean = sumSoFar / samplesSoFar;
    return x - estimatedMean;
  }
}
const normalizedDataset1 =
  unNormalizedDataset1.map(newStreamingZeroMeanFn());
const normalizedDataset2 =
  unNormalizedDataset2.map(newStreamingZeroMeanFn());
```

Возвращает унарную функцию, возвращающую входные значения, из которых вычтено среднее всех входных значений на текущий момент

Обратите внимание, что мы создаем новую функцию отображения, использующую собственные копии счетчика и накопителя элементов. Благодаря этому можно нормализовать несколько наборов данных одновременно. В противном случае оба объекта `Dataset` подсчитывали бы вызовы и суммы с помощью одних и тех же переменных. У этого решения есть свои ограничения, из которых особенно стоит отметить опасность арифметического переполнения переменных `samplesSoFar` и `sumSoFar`, так что осторожность здесь не помешает.

6.2. Обучение моделей с помощью model.fitDataset

Потоковый API `Dataset` очень удобен, и, как мы видели, позволяет выполнять довольно изящные операции над данными, но основная цель API `tf.data` — упрощение подключения к модели источника данных для обучения и оценки работы модели. Но как `tf.data` может помочь в этом?

Начиная с главы 2, мы всегда использовали для обучения модели API `model.fit()`. Как вы помните, этот API принимает на входе два обязательных аргумента — `xs` и `ys`. Напомним также, что переменная `xs` должна быть тензором, представляющим набор входных примеров данных. А переменная `ys` обязана быть тензором, представляющим соответствующий набор выходных целевых признаков. Например, в листинге 5.11 мы обучали и подвергали тонкой настройке модель обнаружения искусственных объектов с помощью таких вызовов:

```
model.fit(images, targets, modelFitArgs)
```

Здесь `images` по умолчанию представлял собой тензор ранга 4 формы `[2000, 224, 224, 3]`, соответствующий набору из 2000 изображений. Объект конфигурации `modelFitArgs` задает размер батча для оптимизатора, по умолчанию равный 128. Как видим, TensorFlow.js получила набор из 2000 примеров данных в оперативной памяти[1] — весь массив данных, после чего прошла в цикле по этим данным по 128 примеров за раз на каждой эпохе.

Но что, если такого количества данных недостаточно и мы хотим обучить модель на гораздо большем наборе данных? В этом случае у нас есть два отнюдь не идеальных варианта. Вариант 1: попытаться загрузить этот намного больший массив данных и посмотреть, что получится. Впрочем, рано или поздно TensorFlow.js не хватит оперативной памяти, и мы получим сообщение об ошибке, указывающее, что выделить память для хранения обучающих данных не удалось. Вариант 2: загружать данные в GPU отдельными порциями и вызывать `model.fit()` для каждой порции отдельно. При этом нам придется самим координировать работу `model.fit()`, итеративно обучая модель на частях обучающих данных по мере их готовности. Чтобы обучать модель в течение более чем одной эпохи, придется возвращаться назад и заново загружать эти порции в определенном (видимо, перетасованном) порядке. Подобная схема работы не только представляется довольно громоздкой и подверженной ошибкам, но и мешает выдаче библиотекой TensorFlow.js корректной информации о количестве эпох и метриках, так что нам самим придется связывать эти показатели воедино.

TensorFlow.js предоставляет намного более удобный инструмент для этой задачи в виде API `model.fitDataset()`:

```
model.fitDataset(dataset, modelFitDatasetArgs)
```

Первым аргументом `model.fitDataset` принимает объект `Dataset`, который должен соответствовать определенному паттерну. А именно, этот объект `Dataset` должен выдавать объекты, содержащие два свойства. Первое свойство — `xs` — типа `Tensor`,

[1] Точнее, в памяти GPU, размер которой обычно еще меньше, чем RAM системы!

содержит признаки для батча примеров данных и аналогично аргументу `xs` метода `model.fit()`, но объект `Dataset` выдает элементы по одному батчу за раз, а не весь массив сразу. Второе обязательное свойство называется `ys` и содержит соответствующий тензор целевых признаков[1]. По сравнению с методом `model.fit()`, метод `model.fitDataset()` обладает несколькими преимуществами. Прежде всего, не нужно писать код для общей организации скачивания частей набора данных — TensorFlow.js берет эту задачу на себя и решает ее эффективным потоковым образом, с получением данных по мере необходимости. Встроенные в объект `Dataset` кэширующие структуры предоставляют возможность упреждающей выборки данных, которые могут понадобиться, экономя таким образом вычислительные ресурсы. Возможности этого вызова API шире и в смысле обучения наборов данных намного большего размера, чем помещается в памяти GPU. Фактически размер набора данных для обучения модели ограничивается теперь лишь имеющимся у нас временем — обучение продолжается до тех пор, пока доступны новые примеры данных. Это поведение иллюстрируется примером data-generator из репозитория tfjs-examples.

В данном примере мы учим модель оценивать вероятность выигрыша в простой азартной игре. Как обычно, для извлечения и запуска примера можно использовать следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/data-generator
yarn
yarn watch
```

Вышеупомянутая игра представляет собой упрощенную версию карточной игры, в чем-то напоминающую покер. Каждый из игроков получает по `N` карт, где `N` — положительное целое число, каждой из которых соответствует случайное целое число от 1 до 13. Правила игры следующие.

- Выигрывает игрок, у которого самая большая группа карт одного достоинства. Например, если у игрока 1 есть три карты одного достоинства, а у игрока 2 — только две, выигрывает игрок 1.
- Если у обоих игроков максимальные группы карт одного достоинства одного размера, то выигрывает игрок с группой карт максимального достоинства. Например, пара пятерок бьет пару четверок.
- Если ни у одного игрока нет даже двух карт одного достоинства, выигрывает игрок с одиночной картой максимального достоинства.
- При равенстве победитель выбирается случайным образом, 50/50.

Как легко убедиться, шансы на выигрыш у обоих игроков одинаковы. Так что, если о картах ничего не известно, угадать, выиграем мы или нет, можно лишь в половине случаев. Мы создадим и обучим модель, которая по картам игрока 1 будет предсказывать, выиграет ли он. На рис. 6.1 видно, что нам удалось добиться без-

[1] В случае моделей с несколькими входными сигналами вместо отдельных тензоров признаков здесь ожидается массив тензоров, аналогично случаю моделей, ориентированных на подгонку под несколько целей.

ошибочности примерно в 75 % для этой задачи после обучения модели на ~250 000 примеров данных (50 эпох × 50 батчей в эпохе × 100 примеров данных в батче). Моделирование проводилось из расчета пяти карт на руки, но подобная точность достижима и при другом количестве. Для более высокой степени безошибочности необходимы бóльшие батчи и большее число эпох, но даже 75 % дают нашему интеллектуальному игроку ощутимое преимущество над «наивным» игроком в оценке вероятности победы.

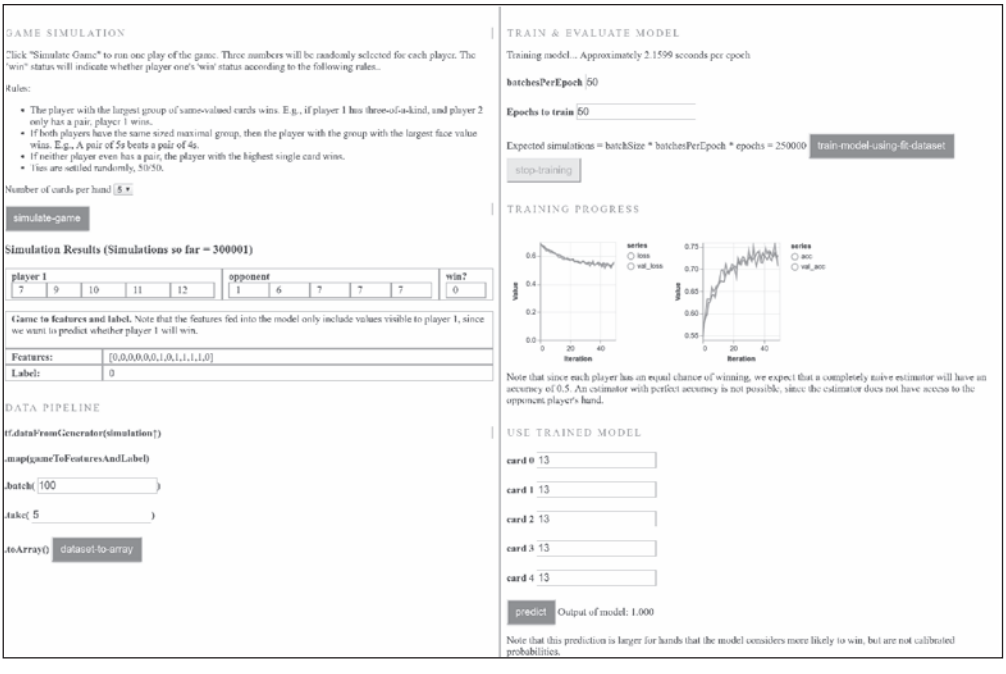


Рис. 6.1. UI примера data-generator

Рассмотрим подробнее скриншот. Описание правил игры и кнопка запуска моделирования находятся слева вверху. Ниже — сгенерированные признаки и конвейер обработки данных. Нажатие кнопки Dataset-to-Array запускает цепочку операций над объектом `Dataset`: моделирование игры, генерацию признаков, организацию примеров данных в батчи, группировку *N* батчей вместе, преобразование их в массив и вывод этого массива в UI. Справа вверху — все, что нужно для обучения модели с помощью этого конвейера обработки данных. При нажатии пользователем кнопки Train-Model-Using-Fit-Dataset начинает выполняться операция `model.fitDataset()`, извлекая примеры данных из конвейера. Под этой кнопкой выводятся кривые потерь и безошибочности. Справа внизу пользователь может ввести значения карт на руках игрока 1 и нажать кнопку для получения от модели соответствующих предсказаний. Чем больше предсказанное значение, тем более модель уверена в победе игрока с такими картами. Выборка значений производится с возвращением в колоду, так что вполне возможны пять одинаковых карт.

Для выполнения этой операции с помощью `model.fit()` нам бы пришлось создать и хранить где-то тензор, содержащий 250 000 примеров данных, для одного только представления входных признаков. Данные в этом примере относительно невелики — всего несколько десятков значений с плавающей точкой на пример[1] — но для нашей задачи обнаружения объектов из предыдущей главы 250 000 примеров данных потребовали бы около 150 Гбайт памяти GPU[2] — намного больше доступного в 2019 году для большинства браузеров объема.

Рассмотрим наиболее интересные части кода. Во-первых, посмотрим на генерацию объекта `Dataset`. Код в листинге 6.7 (упрощенный вариант приведенного в файле `tfjs-examples/data-generator/index.js`) аналогичен генератору объекта `Dataset` для бросания костей из листинга 6.3, но немного сложнее, поскольку мы сохраняем больше информации.

Листинг 6.7. Создание объекта tf.data.Dataset для карточной игры

```
import * as game from './game';

let numSimulationsSoFar = 0;

function runOneGamePlay() {
  const player1Hand = game.randomHand();
  const player2Hand = game.randomHand();
  const player1Win = game.compareHands(
      player1Hand, player2Hand);
  numSimulationsSoFar++;
  return {player1Hand, player2Hand, player1Win};
}

function* gameGeneratorFunction() {
  while (true) {
    yield runOneGamePlay();
  }
}

export const GAME_GENERATOR_DATASET =
   tf.data.generator(gameGeneratorFunction);

await GAME_GENERATOR_DATASET.take(1).forEach(
    e => console.log(e));

// Выводит
// {player1Hand: [11, 9, 7, 8],
// player2Hand: [10, 9, 5, 1],
// player1Win: 1}
```

Библиотека game предоставляет функции randomHand() и compareHands(), для генерации наборов карт на руках для упрощенной карточной игры, похожей на покер, и определения победителя путем сравнения двух таких наборов

Имитация двух игроков в простой карточной игре, похожей на покер

Возвращает два набора карт на руках игроков и победителя

Вычисление победителя

[1] До 2019 года в JavaScript и целые, и дробные числа представлялись с помощью примитивного типа Number. Number, по сути, был числом с плавающей запятой. В 2019 году был предложен тип данных BigInt, который предназначен только для целых чисел. Возможно, в момент написания книги авторы еще не смогли перенять это новшество. — *Примеч. науч. ред.*

[2] числоПримеров × ширинаИзображения × высотаИзображения × глубинаЦвета × размерInt32 = 250 000 × 224 × 224 × 3 × 4 байт.

После подключения нашего простого генератора к игровой логике необходимо отформатировать данные способом, подходящим для нашей задачи обучения. А именно: наша задача состоит в предсказании бита `player1Win` на основе `player1Hand`. Для этого необходимо, чтобы наш объект `Dataset` возвращал элементы формы `[batchOfFeatures, batchOfTargets]`, где признаки вычисляются на основе карт на руках игрока 1. Следующий код (листинг 6.8) — упрощенный вариант кода, приведенного в файле `tfjs-examples/data-generator/index.js`.

Листинг 6.8. Создание набора данных признаков для игроков

```
function gameToFeaturesAndLabel(gameState) {
   return tf.tidy(() => {
    const player1Hand = tf.tensor1d(gameState.player1Hand, 'int32');
    const handOneHot = tf.oneHot(
        tf.sub(player1Hand, tf.scalar(1, 'int32')),
        game.GAME_STATE.max_card_value);

    const features = tf.sum(handOneHot, 0);
    const label = tf.tensor1d([gameState.player1Win]);
    return {xs: features, ys: label};
  });
}

let BATCH_SIZE = 50;

export const TRAINING_DATASET =
  GAME_GENERATOR_DATASET.map(gameToFeaturesAndLabel)
                        .batch(BATCH_SIZE);

await TRAINING_DATASET.take(1).forEach(
    e => console.log([e.shape, e.shape]));

// Выводит форму тензоров:
// [[50, 13], [50, 1]]
```

Получает в качестве аргумента состояние одной сыгранной партии и возвращает представление в виде признаков карт на руках игрока 1 и индикатор победы

Форма тензора handOneHot — [numCards, max_value_card]. Данная операция суммирует количество карт каждого типа, в результате чего получается тензор формы [max_value_card]

Преобразует все элементы из формата выходных объектов игры в массивы двух тензоров: один для признаков, а второй — для цели

Группирует BATCH_SIZE последовательно идущих элементов в один элемент. Также преобразует данные из формата JavaScript-массивов в тензоры, если они не были тензорами изначально

Получив набор данных в нужном виде, можем подключить его к нашей модели с помощью `model.fitDataset()`, как показано в листинге 6.9 (упрощенный вариант кода из файла `tfjs-examples/data-generator/index.js`).

Листинг 6.9. Создание и обучение модели на полученном наборе данных

```
// Формирование модели
model = tf.sequential();
model.add(tf.layers.dense({
  inputShape: [game.GAME_STATE.max_card_value],
  units: 20,
  activation: 'relu'
}));
```

```
model.add(tf.layers.dense({units: 20, activation: 'relu'}));
model.add(tf.layers.dense({units: 1, activation: 'sigmoid'}));

// Обучение модели
await model.fitDataset(TRAINING_DATASET, {
  batchesPerEpoch: ui.getBatchesPerEpoch(),
  epochs: ui.getEpochsToTrain(),
  validationData: TRAINING_DATASET,
  validationBatches: 10,

  callbacks: {
    onEpochEnd: async (epoch, logs) => {
      tfvis.show.history(
          ui.lossContainerElement, trainLogs, ['loss', 'val_loss'])
      tfvis.show.history(
          ui.accuracyContainerElement, trainLogs, ['acc', 'val_acc'],
          {zoomToFitAccuracy: true})
    },
  }
}
```

Данный вызов запускает обучение

Количество батчей в эпохе. Поскольку наш набор данных неограниченный, нужно указать этот параметр, чтобы TensorFlow.js знала, когда выполнять обратный вызов в конце эпохи

В качестве проверочных данных используются обучающие. Обычно так поступать не стоит, чтобы не получить превратное представление о качестве работы модели. В данном случае ничего страшного в этом нет, ведь используемые для обучения и проверки данные заведомо независимы друг от друга благодаря генератору

model.fitDataset() создает совместимую с модулем tfvis «историю обучения» аналогично model.fit()

Необходимо указать TensorFlow.js, сколько примеров данных взять из проверочного набора данных для одной проверки

Как видно из предыдущего листинга, подгонка модели к набору данных оказывается ничуть не сложнее подгонки модели к паре тензоров `x`, `y`. Чтобы все прекрасно работало и мы получали поток данных от (возможно) удаленного источника без каких-либо забот о координации всего процесса, достаточно, чтобы наш объект `Dataset` выдавал тензорные значения в нужном формате. Помимо передачи объекта `Dataset` вместо пары тензоров, у объекта конфигурации есть несколько заслуживающих упоминания отличий.

- `batchesPerEpoch` — как видно в листинге 6.9, в настройках `model.fitDataset()` есть необязательное поле для указания количества батчей в эпохе. Когда мы передавали методу `model.fit()` весь массив данных, можно было легко вычислить, сколько примеров данных во всем наборе данных — просто взять `data.shape[0]`! При использовании же `model.fitDataset()` указать TensorFlow.js, когда заканчивается эпоха, можно двумя способами. Первый способ — воспользоваться указанным полем настроек и `fitDataset()` после соответствующего числа батчей выполнит обратные вызовы `onEpochEnd` и `onEpochStart`. Второй способ — использовать завершение самого набора данных в качестве сигнала его исчерпания. Для моделирования подобного поведения в листинге 6.7 достаточно поменять:

  ```
  while (true) { ... }
  ```

  на:

  ```
  for (let i = 0; i<ui.getBatchesPerEpoch(); i++) { ... }
  ```

- `validationData` при использовании `model.fitDataset()` может также представлять собой набор данных. Но не обязательно. Если хотите, можете продолжать

использовать для `validationData` тензоры. Проверочный набор данных должен удовлетворять тем же спецификациям относительно формата возвращаемых элементов, что и обучающий набор данных.

- `validationBatches` — если проверочные данные берутся из набора данных, необходимо указать TensorFlow.js, сколько именно примеров необходимо взять оттуда для полного набора. Если это значение не задано, TensorFlow.js будет извлекать данные из объекта `Dataset`, пока тот не вернет сигнал об окончании. А поскольку в коде из листинга 6.7 для генерации набора данных используется «вечный» генератор, этого никогда не случится и программа зависнет.

Остальные настройки идентичны соответствующим настройкам API `model.fit()`, так что ничего менять не нужно.

6.3. Распространенные паттерны доступа к данным

Любому разработчику необходимы решения для подключения данных к модели. Спектр таких подключений простирается от распространенных стандартных подключений к широко известным прикладным наборам данных наподобие MNIST до создаваемых для очень узких задач подключений к проприетарным корпоративным форматам данных. В этом разделе мы поговорим о том, как можно упростить создание и сопровождение подобных подключений.

6.3.1. Работаем с форматом данных CSV

Помимо работы с распространенными стандартными наборами данных, доступ к данным чаще всего включает загрузку готовых данных, хранящихся в некоем файловом формате. Файлы данных часто хранятся в формате CSV (значения, разделенные запятыми)[1], так как он простой, удобочитаемый для людей и поддерживается повсеместно. У прочих форматов могут быть свои преимущества по части экономности хранения и скорости доступа, но CSV можно считать своего рода *лингва франка* наборов данных. В сообществе JavaScript обычно необходима возможность удобной потоковой передачи данных из какой-либо конечной HTTP-точки. Именно поэтому TensorFlow.js предоставляет нативную поддержку потоковой обработки и операций над данными из CSV-файлов. В подразделе 6.1.2 мы вкратце описывали создание экземпляра `tf.data.Dataset`, обеспечиваемого данными из CSV-файла. Сейчас заглянем глубже в API CSV и продемонстрируем, насколько модуль `tf.data` упрощает

[1] По состоянию на январь 2019 года сайт конкурсов по науке о данных и машинному обучению kaggle.com/datasets мог похвастаться внушительным списком из 13 971 общедоступного набора данных, из которых 2/3 хранились в формате CSV.

работу с подобными источниками данных. Мы приведем пример приложения, выполняющего подключение к удаленным наборам данных в формате CSV, вывод их схемы, подсчет количества элементов набора данных, а также предоставляющего пользователю возможность выбора и вывода отдельных примеров данных. Получить пример можно с помощью уже хорошо известных вам команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/data-csv
yarn && yarn watch
```

В результате в браузере должен открыться сайт с просьбой ввести URL CSV-файла, расположенного на внешнем сервере, или воспользоваться одним из предлагаемых URL, для чего пользователь может щелкнуть, например, на CSV Boston-housing (рис. 6.2). Под полем для ввода URL находятся кнопки для трех действий: 1) подсчета количества строк в наборе данных, 2) извлечения названий столбцов CSV-файла, если они указаны, и 3) обращения к находящемуся в указанной строке примеру из набора данных и его выводу в UI. Рассмотрим, как эти действия осуществляются и почему API `tf.data` сильно упрощает их реализацию.

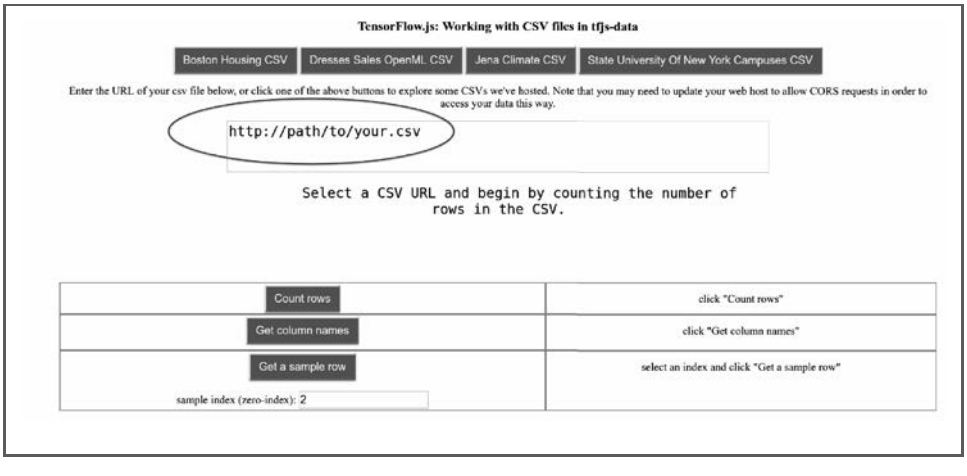


Рис. 6.2. Веб-интерфейс для нашего примера использования CSV. Нажмите одну из кнопок для заранее заданных CSV вверху экрана или введите путь к своему CSV-файлу, расположенному на внешнем сервере, при наличии такового. В последнем случае не забудьте включить доступ по технологии CORS для своего файла

Мы уже видели ранее, что можно очень легко создать набор данных tfjs-data из удаленного CSV-файла с помощью такой команды:

```
const myData = tf.data.csv(url);
```

где `url` представляет собой либо строковый идентификатор, в котором используется протокол http://, https:// или file:// либо объект `RequestInfo`. Данный вызов на самом деле *не* производит никаких запросов к указанному URL для проверки того,

например, существует ли и доступен ли файл, так как выполнение отложенное. В листинге 6.10 CSV-файл сначала извлекается с помощью асинхронного вызова `myData.forEach()`. Вызываемая в `forEach()` функция просто преобразует элементы набора данных в строковый вид и выводит их в UI, но можно представить себе выполнение с помощью этого итератора и других действий, скажем генерации элементов UI для каждого из элементов множества или вычисления сводных показателей для отчета.

Листинг 6.10. Вывод первых десяти записей из удаленного CSV-файла

```
const url = document.getElementById('queryURL').value;
const myData = tf.data.csv(url);
await myData.take(10).forEach(
  x => console.log(JSON.stringify(x))));

// Будет выведено что-то наподобие
// {"crim":0.26169,"zn":0,"indus":9.9,"chas":0,"nox":0.544,"rm":6.023, ...
// ,"medv":19.4}
// {"crim":5.70818,"zn":0,"indus":18.1,"chas":0,"nox":0.532,"rm":6.75, ...
// ,"medv":23.7}
// ...
```

Создаем набор данных tfjs-data путем передачи URL в метод tf.data.csv()

Создаем набор данных, состоящий из первых десяти строк набора данных CSV. Далее проходим в цикле по все значениям этого набора данных с помощью метода forEach(). Учтите, что forEach() — асинхронная функция

В наборах данных формата CSV первая строка часто представляет собой метаданные-заголовок с названиями столбцов. По умолчанию метод `tf.data.csv()` исходит именно из этого предположения, но его поведение можно изменить, передав в качестве второго аргумента объект `csvConfig`. Если в самом CSV-файле названия столбцов не указаны, их можно указать вручную в конструкторе:

```
const myData = tf.data.csv(url, {
    hasHeader: false,
    columnNames: ["firstName", "lastName", "id"]
  });
```

Указанные вручную для набора данных CSV настройки `columnNames` обладают приоритетом перед прочитанной из файла данных строкой-заголовком. По умолчанию набор данных предполагает, что первая строка — строка заголовка. Если же это не так, необходимо явно указать, что заголовок отсутствует, и самостоятельно определить `columnNames`.

После создания объекта `CSVDataset` можно выполнять к нему запросы и получать названия столбцов с помощью метода `dataset.columnNames()`, возвращающего упорядоченный список строковых значений названий столбцов. Метод `dataset.columnNames()` существует только в классе `CSVDataset` и обычно недоступен в объектах `Dataset`, созданных на основе других типов источников. Кнопка Get Column Names в нашем примере связана с обработчиком, использующим как раз этот API. В результате запроса названий столбцов объект `Dataset` выполняет запрос извлечения

к указанному URL для получения первой строки и ее разбора; отсюда и асинхронный вызов в листинге 6.11 (сокращенный вариант кода из `tfjs-examples/csv-data/index.js`).

Листинг 6.11. Получение названий столбцов из CSV

```
const url = document.getElementById('queryURL').value;
const myData = tf.data.csv(url);
    const columnNames = await myData.columnNames();
console.log(columnNames);
// Для набора данных Boston Housing будет выведено что-то вроде
//   [ "crim", "zn", "indus", ..., "tax",
//     "ptratio", "lstat"]
```

Обращение к удаленному CSV-файлу для извлечения и разбора заголовков столбцов

Мы получили названия столбцов. Теперь извлечем одну из строк нашего набора данных. В листинге 6.12 мы покажем, как веб-приложение выводит в UI одну выбранную строку CSV-файла. Какую именно — пользователь выбирает с помощью элемента ввода. Для выполнения этого запроса мы сначала воспользуемся методом `Dataset.skip()`, чтобы создать новый объект `Dataset`, такой же, как и исходный, за исключением пропуска первых `n-1` элементов. А затем создадим с помощью метода `Dataset.take()` объект `Dataset`, заканчивающийся после первого же элемента. И наконец, воспользуемся методом `Dataset.toArray()` для извлечения данных в стандартный JavaScript-массив. Если все в порядке, запрос вернет массив, содержащий ровно один элемент с заданной позиции в наборе данных. Вся эта последовательность действий представлена в листинге 6.12 (сокращенный вариант кода из `tfjs-examples/csv-data/index.js`).

Листинг 6.12. Доступ к выбранной строке из удаленного CSV-источника

sampleIndex — число, возвращаемое элементом UI

Создаем набор данных myData со всеми настройками на чтение из указанного URL, но еще к этому URL не подключенный

Создаем новый объект Dataset, пропуская первые sampleIndex значений

Создаем новый объект Dataset, содержащий лишь первый элемент

```
const url = document.getElementById('queryURL').value;
const sampleIndex = document.getElementById(
    'whichSampleInput').valueAsNumber;
const myData = tf.data.csv(url);
const sample = await myData
                        .skip(sampleIndex)
                        .take(1)
                        .toArray();
console.log(sample);
// Для набора данных Boston Housing будет выведено нечто вроде
// [{crim: 0.3237, zn: 0, indus: 2.18, ..., tax:
// 222, ptratio: 18.7, lstat: 2.94}].
```

Именно в результате этого вызова объект Dataset на самом деле обращается к указанному URL и выполняется извлечение данных. Обратите внимание, что тип возвращаемого значения — массив объектов, в данном случае содержащий ровно один объект, в котором ключи соответствуют названиям заголовков, а значения взяты из соответствующих столбцов

Теперь мы можем взять содержимое строки, представленное — как видно из выведенного функцией `console.log(sample)` в листинге 6.12 (приводится в комментарии) — в виде объекта, содержащего соответствующие названиям столбцов значения, и привести его в вид, подходящий для вставки в наш документ. Учтите: если запросить несуществующую строку, например 400-й элемент объекта `Dataset`, содержащего 300 элементов, будет возвращен пустой массив.

Довольно часто при подключении к удаленным наборам данных используют неправильные URL или неподходящие учетные данные. В подобных случаях лучше перехватить ошибку и вернуть пользователю понятное сообщение об ошибке. А поскольку объект `Dataset` на самом деле не обращается к удаленному ресурсу до тех пор, пока не понадобятся данные, важно расположить код обработки ошибок в правильном месте программы. В листинге 6.13 приведен короткий фрагмент кода обработки ошибок из нашего примера веб-приложения для работы с CSV (сокращенный вариант кода из `tfjs-examples/csv-data/index.js`). Больше подробностей относительно подключения к CSV-файлам, защищенным механизмами аутентификации, см. в инфобоксе 6.2.

Листинг 6.13. Обработка ошибок, возникающих из-за сбоев подключения

```
const url = 'http://some.bad.url';
const sampleIndex = document.getElementById(
    'whichSampleInput').valueAsNumber;
const myData = tf.data.csv(url);
let columnNames;
try {
  columnNames = await myData.columnNames();
} catch (e) {
  ui.updateColumnNamesMessage(`Could not connect to ${url}`);
}
```

Обертывание этой строки в блок try не поможет, ведь извлечение данных по некорректному URL происходит не здесь

На этом этапе будет сгенерирована возникающая из-за сбоя подключения ошибка

В разделе 6.2 мы научились использовать метод `model.fitDataset()`. Мы узнали, что для этого метода необходим набор данных, выдающий элементы строго определенного вида. Напомним, что это объекты с двумя свойствами `{xs, ys}`, где `xs` — тензор, представляющий батч входных примеров данных, а `ys` — тензор, представляющий батч соответствующих целевых признаков. По умолчанию набор данных в формате CSV возвращает элементы в виде JavaScript-объектов, но можно поменять настройки, чтобы он возвращал элементы в виде, более близком к тому, который нужен для обучения модели. Для этого нам придется воспользоваться полем `csvConfig.columnConfigs`[1] метода `tf.data.csv()`. Представьте себе посвященный гольфу CSV-файл, включающий три столбца: *club*, *strength* и *distance*. Для предсказания расстояния по информации о клубе и силе игрока можно применить функцию `map` к исходным выводимым данным для распределения полей по `xs` и `ys`; или, что проще, настроить соответствующим образом средство чтения CSV. Таблица 6.4 демонстрирует, как настроить объект `Dataset` для CSV для разделения свойств признаков и меток и выполнения такой обработки по батчам, чтобы результаты подходили для подачи на вход `model.fitDataset()`.

[1] То есть полем columnConfigs объекта csvConfig. — *Примеч. пер.*

Таблица 6.4. Настройки объекта Dataset для CSV для работы с методом model.fitDataset()

Как устроен и настроен набор данных	Код создания объекта Dataset	Результат dataset.take(1).toArray()[0] (первый возвращаемый объектом Dataset элемент)
Неформатированный CSV, по умолчанию	`dataset = tf.data.csv(csvURL)`	`{club: 1, strength: 45, distance: 200}`
CSV с метками, задаваемыми в columnConfigs	`columnConfigs = {distance: {isLabel: true}}; dataset = tf.data.csv(csvURL, {columnConfigs});`	`{xs: {club: 1, strength: 45}, ys: {distance: 200}}`
CSV с columnConfigs и последующим разбиением по батчам	`columnConfigs = {distance: {isLabel: true}}; dataset = tf.data .csv(csvURL, {columnConfigs}) .batch(128);`	`[xs: {club: Tensor, strength: Tensor}, ys: {distance:Tensor}]` Форма этих трех тензоров = [128]

CSV с columnConfigs и последующим разбиением по батчам и преобразованием из объекта в массив	`columnConfigs = {distance: {isLabel: true}}; dataset = tf.data .csv(csvURL, {columnConfigs}) .map(({xs, ys}) => { return {xs:Object.values(xs), ys:Object.values(ys)}; }) .batch(128);`	`{xs: Tensor, ys: Tensor}`
		Учтите, что функция отображения возвращает записи вида:
		`{xs: [number, number], ys: [number]}`
		Операция разбиения по батчам автоматически преобразует числовые массивы в тензоры. Таким образом, форма первого тензора (xs) = [128, 2]. Форма второго тензора (ys) = [128, 1]

ИНФОБОКС 6.2. Извлечение CSV-данных, защищенных механизмами аутентификации

В предыдущих примерах для подключения к данным, доступным в удаленных файлах, нам достаточно было указать URL. Это очень простой способ, работающий как в Node.js, так и в браузере, но иногда данные защищены аутентификацией и необходимо передать параметры запроса (`Request`). При использовании API `tf.data.csv()` можно вместо URL в виде простой строки передать `RequestInfo`, как показано в следующем листинге. Помимо дополнительного параметра авторизации, в объекте `Dataset` ничего не меняется:

```
> const url = 'http://path/to/your/private.csv'
> const requestInfo = new Request(url);
> const API_KEY = 'abcdef123456789'
> requestInfo.headers.append('Authorization', API_KEY);
> const myDataset = tf.data.csv(requestInfo);
```

6.3.2. Доступ к видеоданным с помощью метода tf.data.webcam()

Один из самых интересных вариантов использования проектов TensorFlow.js — обучение и применение моделей машинного обучения к датчикам мобильных устройств. Выявление движения с помощью встроенного акселерометра мобильного телефона? Понимание звуков или речи с помощью встроенного микрофона? Зрительная помощь с использованием встроенной камеры? Столько замечательных идей, и это лишь начало списка.

В главе 5 мы работали с веб-камерой и микрофоном в контексте переноса обучения. Мы воспользовались камерой для управления Пакманом и микрофоном для тонкой настройки системы распознавания речи. И хотя далеко не для всех типов входных данных существует удобный вызов API, для работы с веб-камерой в `tf.data` простой и удобный API все же есть. Взглянем, что он из себя представляет и как с его помощью выполнять предсказания на основе обученных моделей.

С помощью API `tf.data` можно легко создать итератор для объекта `Dataset`, выдающий поэлементно поток изображений с веб-камеры. В листинге 6.14 приведен простой пример из документации. Первое, что в нем бросается в глаза: вызов метода `tf.data.webcam()`. Этот конструктор принимает в качестве аргумента необязательный HTML-элемент и возвращает итератор для веб-камеры. Конструктор работает только в среде браузера. Если обратиться к этому API в среде Node.js или если веб-камера недоступна, конструктор сгенерирует исключение, указывающее на источник ошибки. Более того, перед открытием веб-камеры браузер запрашивает у пользователя разрешение. Если в разрешении отказано, конструктор также сгенерирует исключение. Добросовестный разработчик должен подготовить для этих случаев понятные пользователю сообщения.

Листинг 6.14. Создание объекта Dataset с помощью tf.data.webcam() и HTML-элемента

Элемент, демонстрирующий видео с веб-камеры и определяющий размер тензора

```
const videoElement = document.createElement('video');
videoElement.width = 100;
videoElement.height = 100;

const webcam = await tf.data.webcam(videoElement);
const img = await webcam.capture();
img.print();
webcam.stop();
```

Конструктор объекта Dataset для видео. videoElement отображает контент с веб-камеры, а также служит для задания размера создаваемых тензоров

Получает один кадр из видеопотока и выдает значение в виде тензора

Останавливает видеопоток и приостанавливает итератор веб-камеры

При создании итератора для веб-камеры важно, чтобы он знал форму генерируемых тензоров. Существует два способа добиться этого. В первом случае, показанном в листинге 6.14, ее задает форма указанного HTML-элемента. Если же необходима другая форма или видео вообще не нужно отображать, можно указать желаемую форму через объект конфигурации, как показано в листинге 6.15. Уч-

тите, что передается значение `undefined` соответствующего HTML-элементу аргумента, так что API создаст скрытый элемент в представлении DOM в качестве дескриптора видео.

Листинг 6.15. Создание простого объекта Dataset для веб-камеры с использованием объекта конфигурации

```
const videoElement = undefined;
const webcamConfig = {
    facingMode: 'user',
    resizeWidth: 100,
   resizeHeight: 100};
const webcam = await tf.data.webcam(
    videoElement, webcamConfig);
```

Создание итератора объекта Dataset для веб-камеры с использованием объекта конфигурации вместо HTML-элемента. Здесь мы также указываем, какую камеру использовать на устройстве с несколькими камерами. 'user' означает фронтальную камеру, в отличие от опции 'environment', соответствующей тыловой камере

С помощью объекта конфигурации можно также кадрировать части видеопотока и менять размер изображения. Сочетая HTML-элемент и объект конфигурации, можно задавать место начала кадрирования и желаемый выходной размер. При этом выходной тензор будет интерполироваться к желаемому размеру. В листинге 6.16 приведен пример выбора прямоугольного фрагмента квадратного видео и уменьшения его размера так, чтобы можно было использовать маленькую модель.

Листинг 6.16. Кадрирование и изменение размера данных, получаемых от веб-камеры

```
const videoElement = document.createElement('video');
videoElement.width = 300;
videoElement.height = 300;

const webcamConfig = {
    resizeWidth: 150,
    resizeHeight: 100,
    centerCrop: true
};

const webcam = await tf.data.webcam(
    videoElement, webcamConfig);
```

Без заданных явным образом настроек размер на выходе определяет videoElement — в данном случае 300 × 300

Пользователь просит вырезать из видео фрагмент 150 × 100

Данные вырезаются из центра видео

Захват данных от этого итератора веб-камеры определяется как HTML-элементом, так и объектом webcamConfig

Важно отметить несколько очевидных различий между подобной разновидностью набора данных и наборами данных, с которыми мы работали до сих пор. Например, веб-камера выдает различные значения в разные моменты времени. А набор данных CSV выдает строки в одном порядке вне зависимости от того, насколько быстро или медленно они извлекаются. Более того, можно получать сколько угодно примеров данных от веб-камеры, пока пользователь их запрашивает. Вызывающая API сторона должна явным образом завершать поток данных, когда больше данных не требуется.

Доступ к данным от итератора веб-камеры производится с помощью метода `capture()`, возвращающего тензор, который отражает последний кадр. Пользователи API могут использовать этот тензор в своей работе, не забывая освобождать выделенную под него память, во избежание утечки. Из-за нюансов асинхронной

обработки данных с веб-камеры лучше производить необходимую предварительную обработку непосредственно захваченного кадра, а не использовать функциональность отложенной обработки `map()` модуля `tf.data`.

Другими словами, вместо обработки данных с помощью `data.map()`:

```
// Плохо:
    let webcam = await tfd.webcam(myElement)
    webcam = webcam.map(myProcessingFunction);
    const imgTensor = webcam.capture();
    // Используем imgTensor здесь
    tf.dispose(imgTensor)
```

лучше применить функцию непосредственно к изображению:

```
// Хорошо:
    let webcam = await tfd.webcam(myElement);
    const imgTensor = myPreprocessingFunction(webcam.capture());
    // Используем imgTensor здесь
    tf.dispose(imgTensor)
```

Не следует использовать методы `forEach()` и `toArray()` для итератора веб-камеры. Для обработки длинных последовательностей полученных с устройства кадров пользователям API `tf.data.webcam()` лучше описать свой собственный цикл с помощью, например, функции `tf.nextFrame()` и вызывать `capture()` с подходящей частотой кадров. Дело в том, что при вызове метода `forEach()` для веб-камеры фреймворк будет извлекать кадры с максимальной частотой, с какой только движок JavaScript браузера способен запрашивать их от устройства. В результате тензоры будут создаваться с частотой, превышающей частоту кадров устройства, что приведет к дублированию кадров и расходу вычислительных ресурсов впустую. По аналогичным причинам итератор веб-камеры *не* следует передавать в качестве аргумента методу `model.fit()`.

В листинге 6.17 показана сокращенная версия цикла предсказания из примера webcam-transfer-learning (Пакман), который мы видели в главе 5. Учтите, что внешний цикл выполняется до тех пор, пока `isPredicting = true`, что определяется элементом UI. А внутри скорость выполнения цикла ограничивается вызовом функции `tf.nextFrame()`, привязанным к частоте обновления UI. Следующий код взят из файла `tfjs-examples/webcam-transfer-learning/index.js`.

Одно последнее примечание: при использовании веб-камеры часто имеет смысл получать, обрабатывать изображение и освобождать выделенную под него память, прежде чем выполнять предсказание. Во-первых, полная обработка изображения моделью гарантирует, что соответствующие весовые коэффициенты модели уже загружены в GPU, а значит, предотвращает возможное подтормаживание в начале работы. Во-вторых, дает аппаратному обеспечению веб-камеры время прогреться и приступить к отправке настоящих кадров. В зависимости от аппаратного обеспечения веб-камеры иногда отправляют пустые кадры до тех пор, пока устройство не прогреется. См. фрагмент кода в листинге 6.18, демонстрирующий реализацию описанной методики в примере webcam-transfer-learning (из файла `webcam-transfer-learning/index.js`).

Листинг 6.17. Использование API tf.data.webcam() в цикле предсказания

```
async function getImage() {
  return (await webcam.capture())
      .expandDims(0)
      .toFloat()
      .div(tf.scalar(127))
      .sub(tf.scalar(1));
}

while (isPredicting) {
  const img = await getImage();

  const predictedClass = tf.tidy(() => {
    // Захват кадра с веб-камеры.

    // Обработка изображения и выполнение предсказания...
     ...

    await tf.nextFrame();
  }
```

Захватывает кадр с веб-камеры и нормализует его к диапазону от –1 до 1. Возвращает батч с изображением (батч из 1 элемента) формы [1, w, h, c]

Переменная webcam здесь ссылается на возвращаемый tfd.webcam итератор; см. init() в листинге 6.18

Получаем следующий кадр из итератора веб-камеры

Ждем следующего кадра перед очередным предсказанием

Листинг 6.18. Создание набора видеоданных из tf.data.webcam()

```
async function init() {
  try {
    webcam = await tfd.webcam(
        document.getElementById('webcam'));
  } catch (e) {
    console.log(e);
    document.getElementById('no-webcam').style.display = 'block';
  }
  truncatedMobileNet = await loadTruncatedMobileNet();

  ui.init();

  // «Прогреваем» модель. Загружаем весовые
  // коэффициенты в GPU и компилируем
  // программы WebGL, чтобы первый сбор данных
  // с веб-камеры происходил без задержек.
  const screenShot = await webcam.capture();
  truncatedMobileNet.predict(screenShot.expandDims(0));
  screenShot.dispose();
}
```

Конструктор для объекта набора видеоданных. Элемент 'webcam' представляет собой элемент video в HTML-документе

Выполняем предсказание для первого полученного от веб-камеры кадра, чтобы убедиться, что модель полностью загружена в аппаратное обеспечение

Возвращаемое webcam.capture() значение представляет собой тензор. Выделенную под него память необходимо впоследствии освободить во избежание утечки

6.3.3. Доступ к аудиоданным с помощью API tf.data.microphone()

Помимо данных изображений, модуль `tf.data` включает специальный API для сбора аудиоданных с аппаратного микрофона. Подобно API веб-камеры, API микрофона создает отложенный итератор, позволяющий вызывающей его программе по мере

необходимости запрашивать кадры, аккуратно упакованные в тензоры, которые подходят для непосредственного использования моделью. Типичный сценарий — сбор аудиокадров для последующего выполнения предсказаний. И хотя технически сгенерировать поток обучающих данных с помощью этого API можно, объединить его с метками — непростая задача.

Листинг 6.19 демонстрирует пример сбора аудиоданных длительностью одна секунда с помощью API `tf.data.microphone()`. Учтите, что при выполнении этого кода браузер попросит у пользователя разрешить доступ к микрофону.

Листинг 6.19. Сбор аудиоданных длительностью одна секунда с помощью API tf.data.microphone()

```
const mic = await tf.data.microphone({
  fftSize: 1024,
  columnTruncateLength: 232,
  numFramesPerSpectrogram: 43,
  sampleRateHz: 44100,
  smoothingTimeConstant: 0,
  includeSpectrogram: true,
  includeWaveform: true
});
const audioData = await mic.capture();
const spectrogramTensor = audioData.spectrogram;
const waveformTensor = audioData.waveform;
mic.stop();
```

Передаваемая в метод microphone конфигурация позволяет управлять некоторыми часто используемыми параметрами аудио. Мы перечислим часть из них в основном тексте

Выполняет захват аудиоданных с микрофона

Спектрограмма аудиоданных возвращается в виде тензора формы [43, 232, 1]

Помимо спектрограммы, можно также извлечь непосредственно сигнал в волновом виде. Форма этих данных: [fftSize * numFramesPerSpectrogram, 1] = [44032, 1]

Для завершения потока аудиоданных и отключения микрофона пользователь должен вызвать stop()

У микрофона есть несколько настраиваемых параметров, позволяющих пользователям контролировать применение к аудиоданным быстрого преобразования Фурье (fast Fourier transform, FFT). Например, чтобы в спектрограмме присутствовало больше или меньше кадров частотного представления аудиоданных либо только определенный диапазон аудиоспектра, скажем лишь частоты, необходимые для слышимой речи. Поля из листинга 6.19 означают следующее.

- `sampleRateHz: 44100`
  - Частота дискретизации волнового сигнала микрофона. Должна быть равна 44 100 или 48 000 и совпадать с частотой самого устройства, в противном случае будет сгенерирована ошибка.
- `fftSize: 1024`
  - Этот параметр определяет количество элементов выборки, на основе которого вычисляется каждый неперекрывающийся «кадр» аудио. Каждый из кадров проходит FFT, и чем больше кадр, тем выше его частотная чувствительность и ниже разрешающая способность по времени, поскольку временна́я информация *внутри кадра* теряется.
  - Обязан быть равен числу в степени 2, от 16 до 8192 включительно. В данном случае `1024` означает вычисление мощности сигнала в полосе частот по выборке примерно из 1024 элементов.

- Учтите, что максимальная измеримая частота равна половине частоты дискретизации, то есть примерно 22 кГц.

- `columnTruncateLength: 232`
  - Определяет долю сохраняемой частотной информации. По умолчанию каждый аудиокадр содержит `fftSize` точек данных, в нашем случае 1024, охватывая таким образом весь спектр частот, от 0 до максимума (22 кГц). Впрочем, чаще всего наибольший интерес представляют низкие частоты. Частоты человеческой речи обычно не превышают 5 кГц, поэтому мы сохраняем только часть данных, соответствующую частотам от 0 до 5 кГц.
  - В данном случае 232 = (5 кГц / 22 кГц) × 1024.
- `numFramesPerSpectrogram: 43`
  - По ряду непересекающихся окон (кадров) вычисляется FFT аудиосэмпла для создания спектрограммы. Данный параметр определяет, сколько именно таких окон учитывается в каждой из возвращаемых спектрограмм. Форма возвращаемой спектрограммы: `[numFramesPerSpectrogram, fftSize, 1]`, то есть `[43, 232, 1]` в нашем случае.
  - Длительность кадра равна `sampleRate/fftSize`. В нашем случае 44 кГц × 1024 составляет примерно 0,023 секунды.
  - Промежутков между кадрами нет, так что длительность спектрограммы в целом составит 43 × 0,023 = 0,98, то есть примерно одну секунду.
- `smoothingTimeConstant: 0`
  - Указывает, насколько нужно смешивать данные предыдущего кадра с текущим. Значение должно быть от 0 до 1.
- `includeSpectogram: true`
  - Если равно `true`, вычисляется и выдается в виде тензора спектрограмма. Если приложению не нужно вычислять спектрограмму (фактически только в случае, когда требуется волновое представление), установите эту опцию в `false`.
- `includeWaveform: true`
  - Если равно `true`, сохраняется и выдается в виде тензора волновое представление сигнала. Можно задать и значение `false`, если это представление вызывающей стороне не требуется. Учтите, что одна из опций `includeSpectogram` и `includeWaveform` должна быть равна `true` и если они обе равны `false`, будет возвращена ошибка. В данном случае мы задали их обе равными `true`, чтобы продемонстрировать возможность подобного варианта, но в большинстве приложений достаточно только одной.

Подобно потоку видеоданных, поток аудиоданных иногда начинается не сразу, и начальные данные с устройства будут бессмысленными. Среди них часто встречаются нули и бесконечности, хотя конкретные значения и длительность «прогрева» зависят от платформы. Оптимальное решение — «прогреть» микрофон в течение короткого промежутка времени, отбросив несколько первых сэмплов, пока не начнут поступать корректные данные. Обычно 200 миллисекунд данных для этого вполне достаточно.

6.4. Вероятно, данные не без изъяна: обработка проблемных данных

Практически наверняка в исходных данных встретятся какие-либо проблемы. Если вы используете свой собственный источник данных и не потратили несколько часов в компании эксперта, анализируя отдельные признаки, их распределения и корреляции, то очень велика вероятность наличия в них изъянов, которые способны испортить вашу модель машинного обучения. Мы, авторы данной книги, утверждаем это со всей уверенностью на основе богатого опыта руководства созданием множества систем машинного обучения во множестве предметных областей, в том числе создания нескольких самостоятельно. Наиболее распространенный симптом этого — отсутствие сходимости модели или сходимость ее к степени безошибочности ниже ожидаемой. Еще один, даже более скверный и непростой для отладки паттерн заключается в том, что модель сходится и демонстрирует неплохие результаты на проверочном и контрольном наборах данных, однако при промышленной эксплуатации работает плохо. Иногда это действительно проблема моделирования или плохое значение гиперпараметра, а может, просто не повезло, но в абсолютном большинстве случаев истинная причина этих проблем — изъян в данных.

Все используемые наборы данных (MNIST, набор данных «Ирисы Фишера» и набор для распознавания речевых команд) мы «за кулисами» вручную просмотрели, избавили от неудачных примеров данных, преобразовали в стандартный удобный формат и подвергли прочим операциям науки о данных, о которых не упоминали. Проблемы с данными могут проявляться во множестве форм, включая отсутствующие значения полей, коррелированные между собой примеры данных и асимметричные распределения. Работа с данными настолько разнообразная и обширная сфера, что можно посвятить ей целую книгу. На самом деле вот одна из них, с намного более полным обзором этих вопросов: *Davis Ashley.* Data Wrangling with JavaScript[1].

Во многих компаниях появились полноценные должности исследователей и администраторов данных. Используемые этими специалистами инструменты и рекомендуемые ими практики очень разнообразны и часто зависят от нюансов конкретной предметной области. В этом разделе мы затронем лишь основы и укажем несколько инструментов, чтобы помочь вам избежать разочарования, когда в результате длительных сеансов отладки моделей оказывается, что проблема была в самих данных. Более подробные сведения о науке о данных вы сможете найти в приводимых нами ссылках на дополнительные источники информации.

6.4.1. Теория данных

Для обнаружения и исправления *плохих* данных сначала необходимо понять, что такое *хорошие* данные. Большая часть теории, лежащей в основе машинного обучения, исходит из допущения, что источником данных является некое *распределение*

[1] Доступна на сайте издательства Manning: https://www.manning.com/books/data-wrangling-with-javascript.

вероятностей. В этой терминологии обучающие данные состоят из набора независимых *примеров данных* (samples). Отдельные примеры данных описываются как пары (x, y), где y — часть примера данных, предсказываемая на основе x. Далее, используемые при выводе данные состоят из набора примеров данных из *точно такого же распределения вероятности, что и обучающие данные*. Единственное важное различие между обучающими данными и данными, используемыми для вывода, состоит в том, что во время вывода модель не видит y. Часть y примера данных необходимо предсказать по части x на основе статистических взаимосвязей, усвоенных моделью из обучающих данных.

Существует множество вариантов, почему реальные данные могут не соответствовать подобному идеалу. Если, скажем, обучающие данные и данные, предназначенные для вывода, взяты из различных распределений, говорят, что набор данных *асимметричен*. Простой пример: если при анализе дорожного трафика на основе таких признаков, как погода и время суток, вы берете все обучающие данные за понедельники и вторники, а контрольные данные — за субботы и воскресенья, то безошибочность модели окажется далеко не идеальной. Распределение дорожного трафика в выходные отнюдь не такое же, как в будни. Еще один пример: представьте себе, что мы создаем систему распознавания лиц, причем обучаем ее на основе набора маркированных данных для своей родной страны. Ничего удивительного, если наша система будет плохо работать в регионах с другими демографическими характеристиками. Большинство проблем асимметрии данных, встречающихся на практике гораздо менее очевидны, чем эти две.

Кроме того, асимметрия в наборе данных может возникнуть и в результате какого-либо резкого изменения при сборе данных. Например, если микрофон сломался посередине сбора обучающего набора данных аудиосэмплов, предназначенного для усвоения голосовых команд, и пришлось купить новый, следует ожидать, что распределение шума и полезного сигнала во второй половине обучающего набора данных будет отличаться от первой. А если во время выполнения вывода для контроля будут использоваться данные только с нового микрофона, то асимметрия возникнет и между обучающим и контрольным наборами данных.

До некоторой степени асимметрия неизбежна. Во многих приложениях обучающие данные были собраны когда-то давно, а приложению передаются текущие данные. Распределение, из которого берутся эти примеры данных, меняется вместе с изменением уклада жизни, интересов людей, моды и прочих факторов. В подобном случае можно только выяснить сущность асимметрии и минимизировать ее влияние. Поэтому многие модели машинного обучения, находящиеся в промышленной эксплуатации, постоянно обучают заново на самых свежих обучающих данных, чтобы не отставать от постоянно меняющихся распределений.

Примеры данных могут также оказаться неидеальными, когда не являются независимыми. Оптимальный вариант — когда примеры данных независимы и одинаково распределены (independent and identically distributed, IID). Но в некоторых наборах один пример каким-либо образом указывает на возможное значение следующего. Примеры данных из такого набора — не независимы. Чаще всего зависимость примеров данных друг от друга возникает вследствие сортировки. Специалистов в области компьютерных наук учат упорядочивать данные

ради скорости доступа к ним и по другим всевозможным причинам. На самом деле системы баз данных часто упорядочивают данные незаметно для пользователей. В результате при потоковой передаче данных из какого-либо источника необходимо соблюдать осторожность, чтобы эти данные не оказались упорядочены каким-либо образом.

Рассмотрим следующий гипотетический пример. Для работы с недвижимостью необходимо оценивать стоимость жилья в Калифорнии. Мы находим набор данных в формате CSV с ценами на жилье[1] со всего штата вместе с соответствующими признаками, например количеством комнат, возрастом здания и т. д. Хочется сразу же приступить к обучению функции, отображающей признаки в цены, раз уж данные у нас есть и мы знаем, как это сделать. Но, как нам хорошо известно, в данных часто встречаются изъяны и имеет смысл сначала внимательнее посмотреть на них. В первую очередь мы построим график зависимости части признаков от их индекса в массиве с помощью объектов `Dataset` и библиотеки Plotly.js (рис. 6.3 и листинг 6.20) (сокращенная версия из https://codepen.io/tfjs-book/pen/MLQOem).

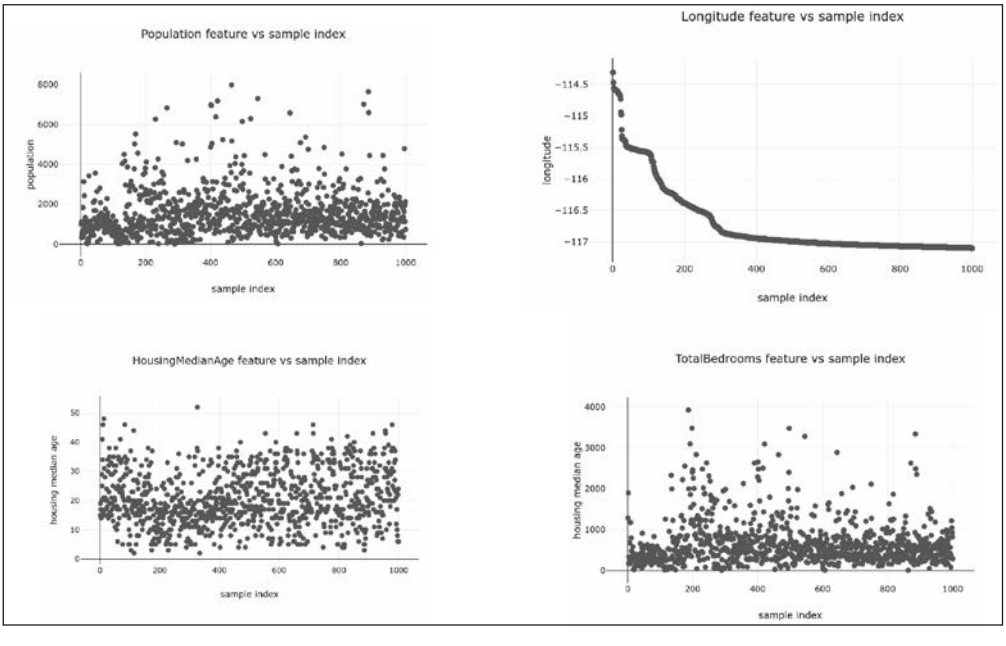


Рис. 6.3. Графики четырех признаков из набора данных относительно индекса примера данных. В идеале в чистом наборе данных IID индекс примера данных не дает никакой информации о значении признака. Здесь же для некоторых признаков распределение значений у явно зависит от x. И что совсем уж никуда не годится, признак «долгота», похоже, отсортирован по индексу примера данных

[1] Описание используемого здесь набора данных цен на жилье в Калифорнии можно найти на сайте экспресс-курса по машинному обучению компании Google: http://mng.bz/Xpm6.

Листинг 6.20. Создание графика признака относительно индекса с помощью модуля tfjs-data

```
const plottingData = {
  x: [],
  y: [],
  mode: 'markers',
  type: 'scatter',
  marker: {symbol: 'circle', size: 8}
};
const filename = 'https://storage.googleapis.com/learnjs-data/csv-
    datasets/california_housing_train.csv';
const dataset = tf.data.csv(filename);
await dataset.take(1000).forEachAsync(row => {
  plottingData.x.push(i++);
  plottingData.y.push(row['longitude']);
});

Plotly.newPlot('plot', [plottingData], {
  width: 700,
  title: 'Longitude feature vs sample index',
  xaxis: {title: 'sample index'},
  yaxis: {title: 'longitude'}
});
```

Собирает значения и индексы первых 1000 примеров данных. Не забудьте ключевое слово await, иначе график (вероятно) окажется пустым!

Представьте себе разбиение подобного набора данных на обучающие и контрольные данные: 500 первых примеров — на обучающие данные, а остальное — на контрольные. Что получится? Исходя из вышеописанного, обучение будет происходить на данных из одной географической области, а контроль — на данных из другой. Блок Longitude (Долгота) на рис. 6.3 демонстрирует суть проблемы: долгота первых примеров данных намного выше (западнее), чем остальных. Вероятно, признаки несут немалый объем полезного сигнала, так что в какой-то мере модель будет «работать», но ни о какой безошибочности или высоком ее качестве, как в случае настоящих IID-данных, говорить не приходится. Если не учесть этого нюанса, можно потратить дни или недели на эксперименты с различными моделями и гиперпараметрами, прежде чем догадаться взглянуть на данные!

Как исправить эту ситуацию? В данном конкретном случае все просто. Чтобы разрушить зависимость между данными и индексом, достаточно лишь перетасовать данные в случайном порядке. Впрочем, нужно учесть один нюанс. У объектов `Dataset` TensorFlow.js есть встроенная *потоковая оконная* функция перетасовки. Это значит, что примеры данных перетасовываются случайным образом в пределах окна фиксированного размера, но не далее. Дело в том, что объекты `Dataset` библиотеки TensorFlow.js производят потоковую обработку данных и поток может состоять из бесконечного числа примеров. Чтобы перетасовать бесконечный источник данных, придется дождаться его завершения.

Можем ли мы обойтись этой потоковой оконной перетасовкой для нашего признака долготы? Конечно, если нам известен размер набора данных (17 000 в данном случае), можно просто задать размер окна больше этого значения, и все в порядке. При очень больших размерах окон оконная перетасовка эквивалентна обычной полной перетасовке. Если же размер набора данных неизвестен или неприемлемо велик

(в том смысле, что набор не помещается полностью в кэше оперативной памяти), придется обходиться меньшим.

Рисунок 6.4, созданный с помощью https://codepen.io/tfjs-book/pen/JxpMrj, иллюстрирует перетасовку данных с использованием метода `shuffle()` объекта `tf.data.Dataset` в случае четырех различных размеров окон:

```
for (let windowSize of [10, 50, 250, 6000]) {
   shuffledDataset = dataset.shuffle(windowSize);
   myPlot(shuffledDataset, windowSize)
}
```

Как видим, структурная зависимость индекса и значения признака остаются очевидными даже для относительно большого размера окна. Только при размере окна 6000 для невооруженного глаза представляется, что данные теперь можно считать IID. Так что, 6000 — как раз подходящий размер окна? Или между 250 и 6000 можно было найти другое подходящее значение? Или 6000 все равно недостаточно, чтобы охватить незаметные на этих иллюстрациях проблемы с распределением? Правильный подход: перетасовка всего набора данных с использованием `windowSize >=` числа примеров данных в наборе. В случае же тех наборов данных, где это невозможно из-за ограничений памяти, временных рамок или потенциальной неограниченности набора данных, вам придется на время стать исследователем данных и, исходя из распределения, определить подходящий размер окна.

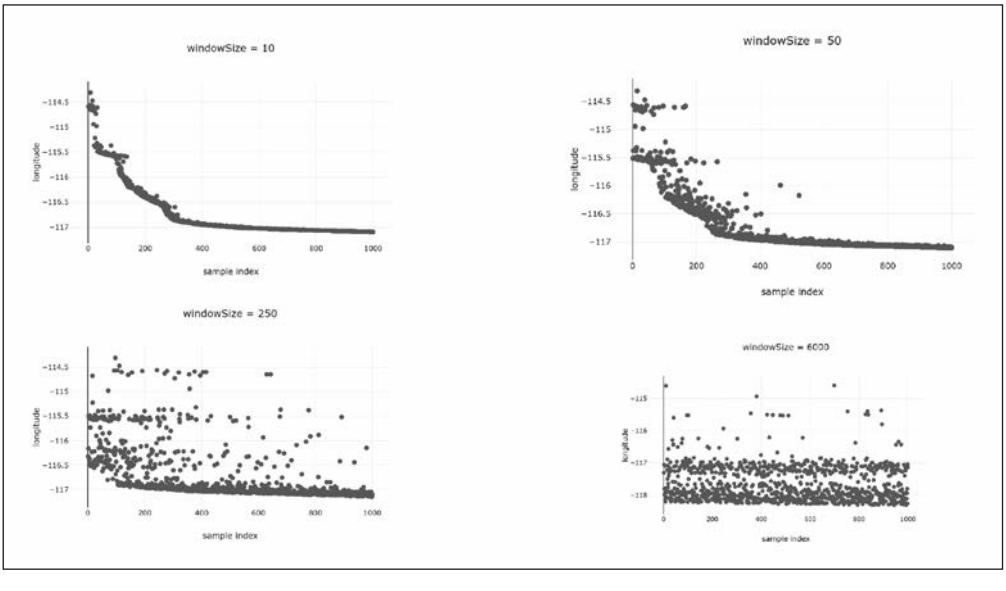


Рис. 6.4. Четыре графика зависимости долготы от индекса примера данных для четырех перетасованных наборов данных. Окна перетасовки у них отличаются: от 10 до 6000 примеров данных. Как видим, даже при размере окна, равном 250 примеров, остается сильная зависимость между индексом и значением признака. Крупных значений больше в начале. Только при размере окна, почти совпадающем с размером набора данных, практически восстанавливается IID-сущность данных

6.4.2. Обнаружение и исправление проблем с данными

В предыдущем разделе мы рассказали, как обнаружить и исправить одну из проблем с данными: зависимость примеров данных друг от друга. Конечно, это лишь одна из множества проблем, связанных с данными. Обсуждение всех возможных сложностей выходит далеко за рамки данной книги, поскольку число проблем в данных ничуть не меньше числа проблем с кодом. Впрочем, рассмотрим здесь некоторые из них, чтобы, столкнувшись с проблемой, вы могли ее узнать и знали, по каким ключевым словам искать дополнительную информацию.

Аномалии

Аномальные значения — очень необычные для конкретного набора примеры данных, иногда не относящиеся к распределению, лежащему в его основе. Например, при работе с набором данных медицинской статистики можно ожидать, что вес типичного взрослого человека находится в диапазоне от 40 до 130 кг. Если в наборе данных 99 % примеров входят в этот диапазон, но изредка встречается бессмысленное значение 145 000 кг, или 0 кг, или, что еще хуже, NaN[1], то подобные значения следует рассматривать как аномальные. Даже поверхностный поиск в Интернете демонстрирует, что существует множество различных мнений о том, как лучше поступать с аномальными значениями. В идеале в наборе данных всего несколько аномальных значений, причем известно, как их найти. Если можно написать программу для отброса аномальных значений, то можно убрать их из набора данных и произвести обучение без них. Конечно, желательно произвести то же самое и во время вывода; иначе возникнет асимметрия. В данном случае можно воспользоваться той же логикой для извещения пользователя о том, что введенный им пример данных является аномальным для системы и необходимо попробовать ввести другой.

Еще один распространенный способ обработки аномальных значений на уровне признаков — ограничивать значения разумным минимумом и максимумом. В нашем случае можно заменить вводимый вес:

```
weight = Math.min(MAX_WEIGHT, Math.max(weight, MIN_WEIGHT));
```

В подобных случаях имеет смысл также добавить новый признак, указывающий, что аномальное значение было заменено. Благодаря этому можно будет отличить настоящее значение 40 кг от значения –5 кг, которое было заменено на 40, благодаря чему сеть сможет усвоить зависимость между состоянием аномального значения и целевым признаком при наличии таковой:

```
isOutlierWeight = weight > MAX_WEIGHT | weight < MIN_WEIGHT;
```

[1] Ввод значения NaN во входных признаках приведет к распространению этого NaN по всей модели.

Недостающие данные

Зачастую встречаются ситуации, когда в некоторых примерах данных отсутствует часть признаков. Подобное может происходить по многим причинам. Источником данных могут служить вводимые вручную формы, в которых отдельные поля просто пропущены пользователем. Или датчики могут не работать либо быть отключены во время сбора данных. Для некоторых признаков определенные значения просто не имеют смысла. Например, какова последняя цена продажи для дома, который еще никогда не продавался? Или номер телефона для человека, у которого нет телефона?

Как и в случае аномальных значений, существует множество способов решения проблемы с недостающими данными, и мнения исследователей данных о том, какие методики лучше применять в каких случаях, также сильно разнятся. Оптимальная методика зависит от нескольких соображений, включая то, зависит ли вероятность отсутствия признака от значения самого признака или от того, возможно ли спрогнозировать отсутствие значения на основе прочих признаков в примере данных. В инфобоксе 6.3 приведен перечень различных категорий отсутствующих данных.

ИНФОБОКС 6.3. Категории отсутствующих данных

Случайные пропуски (missing at random, MAR).

- Вероятность отсутствия признака не зависит от скрытого отсутствующего значения, но может зависеть от какого-либо другого наблюдаемого значения.
- Пример: автоматизированная система обработки визуальных данных для автомобильного трафика может, помимо прочего, фиксировать регистрационные номера автомобилей и время суток. Иногда, в темноте, не удается распознать регистрационный номер. Наличие признака номера не зависит от его значения, но может зависеть от (наблюдаемого) признака времени суток.

Совершенно случайные пропуски (missing completely at random, MCAR).

- Вероятность отсутствия признака не зависит ни от скрытого отсутствующего значения, ни от какого-либо другого наблюдаемого значения.
- Пример: космическое излучение создает помехи для оборудования и иногда портит значения наборов данных. Вероятность такой порчи не зависит ни от хранимого значения, ни от прочих значений набора данных.

Неслучайные пропуски (missing not at random, MNAR).

- Вероятность отсутствия признака зависит от скрытого значения при заданных наблюдаемых данных.
- Пример: персональные метеостанции отслеживают разнообразную статистику, например информацию об атмосферном давлении, осадках и уровне солнечного излучения. Однако во время снегопада датчик солнечного излучения не принимает сигнала.

При отсутствии каких-либо данных в обучающем наборе приходится вносить исправления, чтобы можно было преобразовать данные в тензор фиксированной

формы, требующий наличия значения в каждой ячейке. Существует четыре основных методики решения проблемы отсутствующих данных.

Простейший из этих методов, применимый в том случае, когда обучающих данных много, а пропущенные поля — редкость, состоит в отбросе тех обучающих примеров данных, в которых отсутствуют какие-либо данные. Впрочем, учтите, что при использовании этого метода можно ненароком внести в модель систематическую ошибку. Для большей наглядности представьте себе задачу, в которой отсутствующие данные из позитивного класса встречаются намного чаще, чем из негативного. В результате модель усвоит неправильные вероятности классов. Спокойно отбрасывать примеры данных можно лишь тогда, когда отсутствующие данные относятся к категории MCAR.

Листинг 6.21. Решение проблемы отсутствующих признаков путем удаления данных

```
const filteredDataset =
    tf.data.csv(csvFilename)
   .filter(e => e['featureName']);
```

Оставляет только элементы, значение 'featureName' которых истинно: то есть не равно 0, null, undefined, NaN или пустой строке

Еще одна методика решения проблемы отсутствия данных, известная как *подстановка* (imputation), — заполнение пропущенных полей каким-либо значением. В числе распространенных методов подстановки — замена отсутствующих значений числовых признаков средним, медианным или модальным значением этого признака. Вместо отсутствующих категориальных признаков можно указать чаще всего встречающееся значение (моду). Более сложные методики включают создание и использование предикторов для отсутствующих признаков на основе имеющихся признаков. Фактически нейронные сети представляют собой одну из таких «сложных методик» подстановки отсутствующих данных. Недостаток подстановки состоит в том, что обучаемая модель не знает, что признак отсутствовал. Если само отсутствие значения несло какую-либо информацию о целевой переменной, то при подстановке эта информация теряется.

Листинг 6.22. Решение проблемы отсутствующих признаков с помощью подстановки

```
async function calculateMeanOfNonMissing(
    dataset, featureName) {
  let samplesSoFar = 0;
  let sumSoFar = 0;
  await dataset.forEachAsync(row => {
    const x = row[featureName];
    if (x != null) {
      samplesSoFar += 1;
      sumSoFar += x;
    }
  });
  return sumSoFar / samplesSoFar;
}

function replaceMissingWithImputed(
    row, featureName, imputedValue)) {
  const x = row[featureName];
```

Функция вычисления используемого для подстановки значения. Имейте в виду: при вычислении среднего значения следует учитывать только корректные значения

Отсутствующими здесь считаются как значения undefined, так и null. В некоторых наборах данных для указания на отсутствие значения могут использоваться специальные значения-индикаторы. Внимательно изучите данные, с которыми работаете!

Учтите, что если вообще все данные отсутствуют, эта функция вернет NaN

Функция для обновления строки по условию, на случай, если значение для featureName отсутствует

```
  if (x == null) {
    return {...row, [featureName]: imputedValue};
  } else {
    return row;
  }
}

const rawDataset tf.data.csv(csvFilename);
const imputedValue = await calculateMeanOfNonMissing(
    rawDataset, 'myFeature');
const imputedDataset = rawDataset.map(
    row => replaceMissingWithImputed(
        row, 'myFeature', imputedValue));
```

Для отображения замены на все нужные элементы используется метод map() объекта tf.data.Dataset

Иногда вместо отсутствующих значений указывается *значение-индикатор* (sentinel value). Например, отсутствующее значение веса человека может заменяться значением –1, указывающим, что вес не измерялся. В подобном случае не забудьте обработать значение-индикатор *до* того, как оно окажется заменено аномальным (например, –1 будет, как мы описывали ранее, изменено на 40).

Теоретически, если между отсутствием признака и предсказываемым целевым признаком существует зависимость, модель может успешно использовать значение-индикатор. На практике модель потратит часть вычислительных ресурсов на усвоение того, когда признак используется в качестве значения, а когда — в качестве индикатора.

Вероятно, наиболее надежный способ решения проблемы с отсутствующими данными — сочетание подстановки для заполнения значения и второго признака-индикатора, указывающего модели на отсутствие признака. В нашем случае мы могли бы заменить отсутствующее значение веса неким предположительным значением и добавить новый признак `weight_missing`, равный 1 при отсутствии значения веса и 0 при его наличии. Благодаря этому модель сможет полноценно учесть отсутствие значения, если это имеет смысл, не путая его с фактическим значением веса.

Листинг 6.23. Добавление признака для индикации отсутствия значения

```
function addMissingness(row, featureName)) {
  const x = row[featureName];
  const isMissing = (x == null) ? 1 : 0;
  return {...row, [featureName + '_isMissing']: isMissing};
}

const rawDataset tf.data.csv(csvFilename);
const datasetWithIndicator = rawDataset.map(
      (row) => addMissingness(row, featureName);
```

Функция для добавления нового признака во все строки, равного 1 при отсутствии значения и 0 в ином случае

Для отображения дополнительного признака на все строки используется метод map() объекта tf.data.Dataset

Асимметрия

Ранее в этой главе мы описывали понятие асимметрии — различия распределений в разных наборах данных. Это одна из главных проблем, встречающихся специалистам по машинному обучению при развертывании обученных моделей для про-

мышленной эксплуатации. Обнаружение асимметрии требует моделирования распределений наборов данных и их сравнения. Простейший способ быстро получить статистические показатели набора данных — воспользоваться утилитой наподобие Facets (https://pair-code.github.io/facets/) (рис. 6.5). Facets анализирует набор данных и выводит его сводные показатели, отображая распределения для всех признаков, позволяя пользователю быстро выявить потенциальные проблемы различия распределений в своих наборах данных.

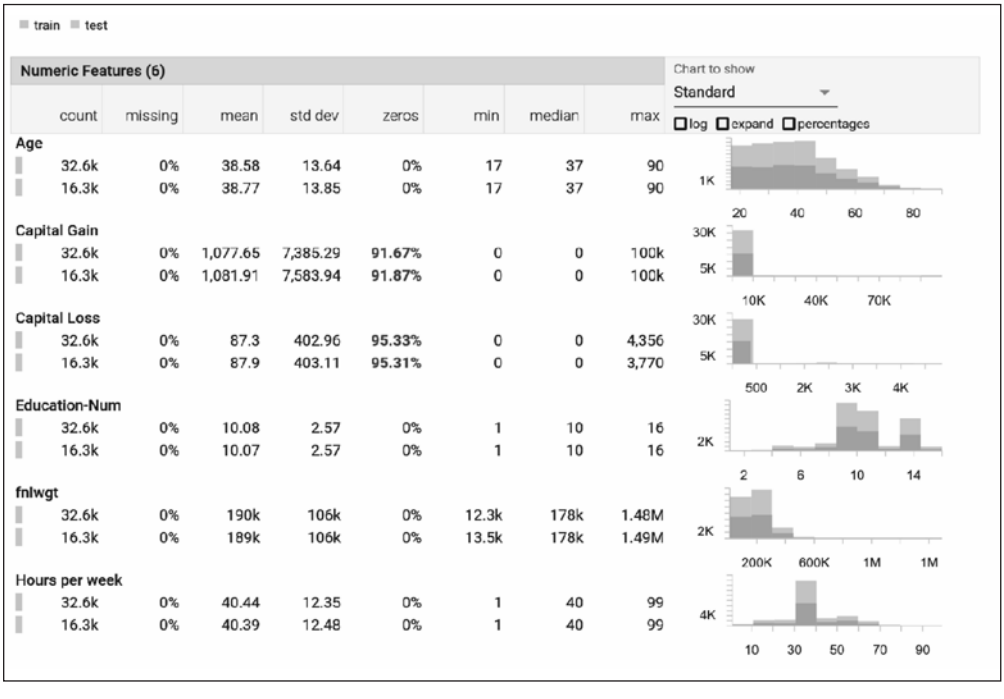


Рис. 6.5. Снимок экрана Facets, демонстрирующий распределения значений по признакам обучающего и контрольного наборов для набора данных UC Irvine Census Income (http://archive.ics.uci.edu/ml/datasets/Census+Income). Этот набор загружен по умолчанию по адресу https://pair-code.github.io/facets/, но вы можете перейти на сайт и загрузить свои собственные CSV-файлы для сравнения. Данное визуальное представление называется Facets Overview

Простейший алгоритм обнаружения асимметрии состоит в вычислении среднего значения, медианы и дисперсии признаков и сравнении их для имеющихся наборов данных — не выходят ли они за рамки допустимого. В более сложных методах делается попытка предсказания, к какому набору относится заданный пример данных. В идеале это должно быть невозможно, поскольку все примеры данных должны относиться к одному распределению. Успешное предсказание того, относится точка данных к обучающему или контрольному набору данных, — признак асимметрии.

Испорченные строковые значения

Категориальные данные часто поступают в виде строковых признаков. Например, при журналировании обращений к веб-странице может фиксироваться используемый браузер с помощью значений вида `FIREFOX`, `SAFARI` и `CHROME`. Обычно такие значения перед вводом в модель глубокого обучения преобразуются в целочисленные (в соответствии с заданным словарем или с помощью хеширования), после чего отображаются в многомерное векторное пространство (см. посвященный вложениям слов подраздел 9.2.3). При этом часто возникает проблема из-за того, что формат строковых значений из одного набора данных отличается от строковых значений другого. Например, в обучающих данных встречается значение `FIREFOX`, а при выполнении вывода модель получает значение `FIREFOX\n`, включающее символ новой строки, или `"FIREFOX"`, с кавычками. Это особенно коварная разновидность асимметрии, к которой желательно отнестись с особым вниманием.

Прочие нюансы данных, которые желательно учитывать

Помимо перечисленных в предыдущих разделах проблем, при вводе данных в систему машинного обучения следует обратить внимание еще на несколько нюансов.

- *Слишком несбалансированные данные* — желательно избавляться от признаков, принимающих одно и то же значение почти для всех примеров данных в наборе. Подобная разновидность сигнала может легко привести к переобучению, а методы глубокого обучения плохо подходят для работы с очень разреженными данными.
- *Различение числовых/категориальных данных* — в некоторых наборах данных для элементов перечислимых множеств используются целые числа, что может привести к проблемам, если на самом деле эти элементы не обладают соответствующей упорядоченностью. Например, при наличии перечислимого множества музыкальных жанров (`ROCK`, `CLASSICAL` и т. д.) и ассоциативного массива соответствий этих жанров целым числам важно обращаться с этими числами при передаче в модель как со значениями перечислимого типа, то есть кодировать их с помощью унитарного представления или вложения (см. главу 9). В противном случае они будут интерпретироваться как значения с плавающей точкой, что предполагает не существующие на самом деле зависимости между термами в соответствии с численным расстоянием между их представлениями.
- *Отличия масштабов* — хотя мы уже упоминали эту разновидность, не помешает рассказать о ней снова, в подразделе, посвященном проблемам с данными. Остерегайтесь числовых признаков с различными масштабами величин. Их наличие может привести к неустойчивости при обучении. В общем случае лучше перед обучением проводить нормализацию по z-оценке (нормализовать среднее значение и среднеквадратичное отклонение) данных. Главное, не забудьте выполнять такую же обработку данных при выполнении вывода, как и перед обучением. См. пример классификации ирисов из репозитория tensor-flow/tfjs-examples, обсуждавшийся в главе 3.

- *Систематическая ошибка, безопасность и защита персональной информации* — разумеется, добросовестная разработка систем машинного обучения включает в себя намного больше, чем можно описать в одной главе книги. Если вы разрабатываете решения на основе машинного обучения, не пожалейте времени на знакомство хотя бы с основами рекомендуемых практик, касающихся систематической ошибки, безопасности и защиты персональной информации. Для начала можете заглянуть на посвященную добросовестным практикам ИИ страницу https://ai.google/education/responsible-ai-practices. Ответственному специалисту очень важно следовать этим практикам. Кроме того, даже из чисто корыстных соображений не помешает учитывать указанные нюансы, ведь даже небольшая систематическая ошибка или проблемы с безопасностью и защитой персональной информации порой приводят к досадным системным сбоям, после которых заказчики начинают искать другие, более надежные решения.

Как правило, лучше потратить немного времени, но убедиться, что данные соответствуют ожидаемому. Для упрощения этой задачи существует множество утилит, от блокнотов Observable, Jupyter, Kaggle Kernel и Colab до утилит с графическим интерфейсом наподобие Facets. На рис. 6.6 приведен еще один способ исследования данных в Facets. В данном случае для просмотра точек из набора данных университетов штата Нью-Йорк (State Universities of New York, SUNY) мы воспользовались средством построения графиков Facets — Facets Dive. С помощью Facets Dive пользователи могут выбирать столбцы данных и отображать их визуально, настраивая отображение под свои потребности. В данном случае мы воспользовались выпадающими меню, где поле Longitude1 служит для x-координаты точки, поле Latitude1 — для y-координаты точки, строковое поле City — для названия точки и Undergraduate Enrollment — для ее цвета. Можно ожидать, что график широты и долготы на двумерной плоскости будет соответствовать карте штата Нью-Йорк, и так оно и получается. Правильность этой карты можно проверить, сравнив ее с веб-страницей SUNY по адресу http://www.suny.edu/attend/visit-us/campus-map/.

6.5. Дополнение данных

Итак, мы собрали данные, подключили их к объекту `tf.data.Dataset` для упрощения операций над ними, а также тщательно их проверили и очистили от всех возможных проблем. Что еще необходимо сделать для успешной работы модели?

Иногда данных оказывается недостаточно и приходится расширять набор программным образом, создавая новые примеры данных путем небольших изменений уже существующих данных. Например, вспомним задачу классификации рукописных цифр MNIST из главы 4. Набор данных MNIST содержит 60 000 обучающих изображений десяти рукописных цифр, по 6000 на цифру. Достаточно ли этого для усвоения всех видов написания, которые должен распознавать наш классификатор? Что, если кто-то напишет слишком большую или маленькую цифру? Или слегка наклоненную? Или перекошенную? Или ручкой с более тонким/толстым пером? Сможет ли наша модель распознать ее?

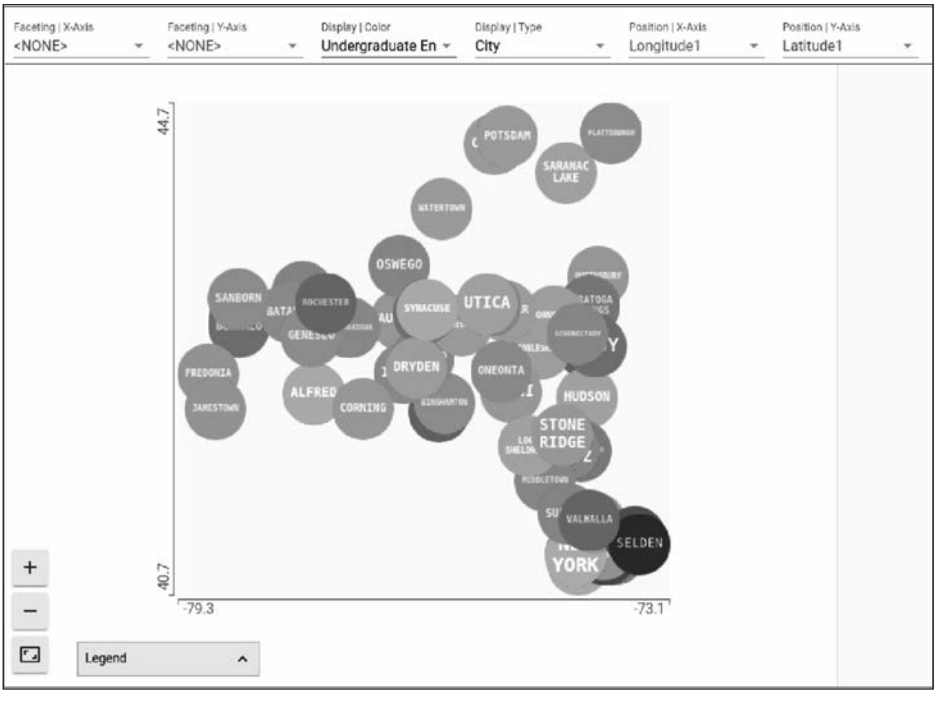


Рис. 6.6. Еще один снимок экрана Facets, на этот раз для набора данных кампусов университетов штата Нью-Йорк из примера data-csv. На нем приведено представление Facets Dive, с помощью которого можно исследовать отношения различных признаков набора данных. Каждая из показанных здесь точек представляет собой точку данных из набора, причем мы настроили представление так, что x-координата точки соответствует признаку Longitude1, y-координата — признаку Latitude1, цвет отражает признак Undergraduate Enrollment, а написанные сверху слова — признак City, содержащий название города, в котором находится кампус университета для каждой из точек данных. Видно, что визуализация приблизительно повторяет очертания штата Нью-Йорк, с Буффало на западе и Нью-Йорком на юго-востоке. При этом один из самых больших кампусов по числу студентов — в городе Селден

Если взять пример цифры из MNIST и изменить изображение, сдвинув цифру на один пиксел влево, семантическая метка цифры не изменится. Сдвинутая влево 9 остается 9, но получается новый обучающий пример. Подобный программно сгенерированный пример данных, созданный путем видоизменения существующего примера, называется *псевдопримером данных* (pseudo-example), а процесс добавления псевдопримеров данных — *дополнением данных* (data augmentation).

Дополнение данных заключается в генерации дополнительных данных на основе уже существующих обучающих примеров. В случае данных изображений в результате таких преобразований, как поворот, кадрирование и масштабирование, нередко получаются правдоподобно выглядящие изображения. Цель всего этого — повышение разнообразия обучающих данных ради расширения возможностей обобщения обученной модели (другими словами, для снижения степени переобучения), что особенно полезно при небольшом размере набора данных.

На рис. 6.7 дополнение данных применяется к входному примеру данных, представляющему собой изображение котенка, из набора маркированных изображений. Дополнение данных производится путем преобразований в отношении этого изображения: вращения и наклона таким образом, что метка примера данных (CAT) не меняется, но сам входной пример меняется существенно.

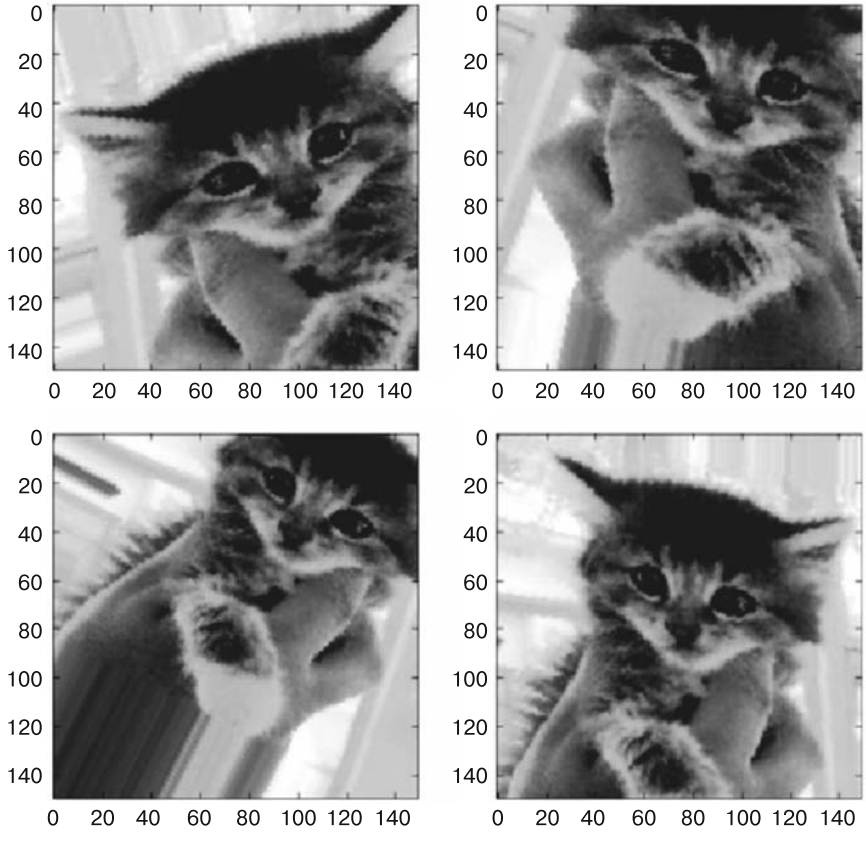


Рис. 6.7. Генерация изображений котенка путем произвольного дополнения данных. Из единственного маркированного примера данных получается целое семейство обучающих примеров, за счет произвольных вращений, отражений, сдвигов и наклонов. Мяу

При использовании подобной схемы дополнения данных для обучения один и тот же сигнал никогда не подается на вход новой сети дважды. Но ее входные сигналы тем не менее сильно коррелируют между собой, поскольку их источник — небольшой набор исходных изображений, ведь мы не можем подобным образом создать новую информацию, а можем только перекраивать уже существующую. Таким образом этого не всегда достаточно для полного избавления от переобучения. Кроме того, при дополнении данных возникает риск того, что распределение обучающих данных теперь может не совпасть с распределением данных, используемых для

вывода, в результате чего возникает асимметрия. Перевешивает ли польза от дополнительных обучающих псевдопримеров данных недостатки асимметрии, зависит от конкретного приложения, и иногда единственный выход — экспериментировать и проверять.

Листинг 6.24 демонстрирует возможность реализации дополнения данных в виде функции `dataset.map()` для внедрения допустимых преобразований в набор данных. Учтите, что дополнение необходимо применять к каждому примеру данных отдельно. Важно также понимать, что дополнение *не следует* применять к проверочному и контрольному наборам. Контроль работы модели на дополненных данных приведет к искаженной мере качества модели, поскольку при выполнении вывода никакого дополнения данных не будет.

Листинг 6.24. Обучение модели на дополненном наборе данных

```
function augmentFn(sample) {
  const img = sample.image;
  const augmentedImg = randomRotate(
      randomSkew(randomMirror(img)));
  return {image: augmentedImg, label: sample.label};
}

const (trainingDataset, validationDataset} =
    getDatsetsFromSource();
augmentedDataset = trainingDataset
   .repeat().map(augmentFn).batch(BATCH_SIZE);

// Обучение модели
await model.fitDataset(augmentedDataset, {
  batchesPerEpoch: ui.getBatchesPerEpoch(),
  epochs: ui.getEpochsToTrain(),
  validationData: validationDataset.repeat(),
  validationBatches: 10,
  callbacks: { ... },
  }
}
```

Функция дополнения данных получает пример данных в формате {изображение, метка} и возвращает новый, измененный пример данных в том же формате

Считаем, что функции randomRotate, randomSkew и randomMirror описаны где-то в другой библиотеке. Коэффициент вращения, наклона и т. д. генерируется случайным образом при каждом вызове. Дополнение должно зависеть только от признаков, а не от метки примера данных

Эта функция возвращает два объекта tf.data.Dataset с типом элементов {изображение, метка}

Дополнение применяется к отдельным элементам, перед организацией в батчи

Подгоняем модель на дополненном наборе данных

Важно! Не применяйте дополнение данных к проверочному набору. Поскольку данные не обходятся в цикле автоматически, здесь для validationData вызывается метод repeat. В соответствии с настройками для каждой проверки берется только десять батчей

Надеемся, эта глава убедила вас, насколько важно разобраться в своих данных, прежде чем подавать их на вход моделей машинного обучения. Мы поговорили о готовых инструментах наподобие Facets, с помощью которых можно исследовать наборы данных, а значит, углублять свои знания о них. Впрочем, для более гибких и подогнанных под нужды пользователей визуализаций данных приходится писать код. В следующей главе мы обучим вас основам tfjs-vis — модуля визуализации, созданного и поддерживаемого авторами TensorFlow.js, подходящего для подобных сценариев визуализации данных.

Упражнения

1. Расширьте пример simple-object-detection из главы 5 так, чтобы использовать `tf.data.generator()` и `model.fitDataset()` вместо генерации заранее всего набора данных. Каковы преимущества подобного варианта? Меняется ли существенно качество работы модели, если ей предоставить для обучения намного больший набор изображений?
2. Произведите дополнение данных примера MNIST за счет применения небольших сдвигов, масштабирований и поворотов к примерам данных. Улучшается ли работа модели? Имеет ли смысл проводить проверку на дополненном потоке данных или лучше будет выполнять контроль только на «настоящих» исходных примерах данных?
3. Постройте графики части признаков из использовавшихся в других главах наборов данных с помощью приведенных в подразделе 6.4.1 методик. Настолько ли данные независимы друг от друга, как вы ожидали? Встречаются ли аномальные значения? А отсутствующие значения?
4. Загрузите какие-нибудь из обсуждавшихся здесь наборов данных в формате CSV в утилиту Facets. Какие признаки, на ваш взгляд, могут вызвать проблемы? Есть какие-нибудь неожиданности?
5. Взгляните на использовавшиеся в предыдущих главах наборы данных. Какие методики дополнения данных подойдут для них?

Резюме

- Данные — главное топливо революции глубокого обучения. Без доступа к большим, хорошо организованным наборам данных большинство приложений глубокого обучения просто невозможно.
- В состав TensorFlow.js входит API `tf.data`, упрощающий потоковую обработку больших наборов данных, разнообразные преобразования данных и подключение их к моделям для обучения и предсказания.
- Существует несколько способов создания объекта `tf.data.Dataset`: из JavaScript-массива, из CSV-файла и на основе функции генерации данных. Чтобы создать объект `Dataset` для потоковой передачи данных из удаленного CSV-файла, достаточно одной строки кода.
- У объектов `tf.data.Dataset` есть API, допускающие организацию цепочкой, что значительно упрощает перетасовку, фильтрацию, организацию в батчи, отображение и прочие операции, часто требуемые в приложениях машинного обучения.
- Объекты `tf.data.Dataset` осуществляют потоковый отложенный доступ к данным, упрощая работу с большими удаленными наборами данных и повышая ее эффективность, правда, это требует использования асинхронных операций.

- Объекты `tf.Model` можно обучать непосредственно из `tf.data.Dataset` с помощью их метода `fitDataset()`.
- Проверка корректности и очистка данных требует времени и терпения, но жизненно необходима для любой системы машинного обучения, применяемой на практике. Обнаружение и исправление таких проблем, как систематическая ошибка, отсутствующие данные и аномальные значения на этапе обработки данных, экономит время на отладку на этапе моделирования.
- Дополнение данных применяется для расширения набора псевдопримерами данных, генерируемыми программным образом, и позволяет модели охватить известные варианты данных, недостаточно представленные в исходном наборе.

7 Визуализация данных и моделей

В этой главе

- Применение tfjs-vis для настраиваемых визуализаций данных.
- Получаем полезную информацию, заглядывая внутрь моделей после обучения.

Визуализация — важный навык для любого специалиста-практика в области машинного обучения, ведь она требуется на всех этапах технологического процесса ML. Данные изучаются с помощью визуализации перед созданием модели; во время проектирования и обучения модели проводится мониторинг процесса обучения посредством визуализации; затем визуализация позволяет понять, как работает уже обученная модель.

В главе 6 мы рассказали вам о преимуществах, которых можно добиться за счет визуализации и понимания данных до применения к ним машинного обучения. Мы описали использование Facets — браузерной утилиты, с помощью которой можно быстро и интерактивно исследовать данные. В этой главе мы познакомим вас с новой утилитой, tfjs-vis, позволяющей визуализировать данные программным образом, настраивая визуализацию под свои нужды. Преимущество такого способа, по сравнению с просмотром исходных данных или использованием готовых инструментов вроде Facets, состоит в более гибкой и универсальной парадигме, ведущей к более глубокому пониманию данных.

Помимо самой визуализации данных, мы покажем, как использовать ее для моделей глубокого обучения *после* их обучения. В нескольких захватывающих примерах заглянем внутрь «черных ящиков» нейронных сетей с помощью визуализации их

внутренних функций активации и вычисления паттернов, сильнее всего «возбуждающих» слои сверточной сети. И тем самым завершим рассказ о тесном взаимодействии визуализации и глубокого обучения на всех его этапах.

К концу главы вы уже будете знать, почему визуализация — незаменимая часть любого технологического процесса машинного обучения. А также познакомитесь со стандартными способами визуализации данных и моделей в фреймворке TensorFlow.js и при необходимости сумеете применить их в собственных задачах машинного обучения.

7.1. Визуализация данных

Начнем с визуализации данных, ведь именно к ней первым делом приступает любой практикующий специалист по машинному обучению, столкнувшись с новой задачей. Допустим, наша задача визуализации выходит за рамки возможностей Facets (например, ее данные не хранятся в небольшом CSV-файле). Сначала мы познакомим вас с простейшим API построения диаграмм, с помощью которого можно создавать в браузере простые и широко используемые типы графиков, включая линейные диаграммы, диаграммы рассеяния, столбчатые диаграммы и гистограммы. А после простых примеров на зашитых в код данных перейдем к примеру визуализации интересного реального набора данных.

7.1.1. Визуализация данных с помощью tfjs-vis

tfjs-vis — библиотека визуализации, тесно интегрированная с TensorFlow.js. В числе множества ее возможностей, описанных в этой главе, есть и облегченный API построения диаграмм, расположенный в ее пространстве имен `tfvis.render.*`[1]. С помощью этого простого и интуитивно понятного API можно строить диаграммы в браузере с упором на чаще всего используемые в машинном обучении виды графиков. Чтобы познакомить вас с `tfvis.render`, мы подготовили его обзор на CodePen по адресу https://codepen.io/tfjs-book/pen/BvzMZr, демонстрирующий использование `tfvis.render` для создания простых типов визуального представления данных.

Основы tfjs-vis

Во-первых, обратите внимание, что библиотека tfjs-vis отделена от основной библиотеки TensorFlow.js. Это видно из того, как CodePen импортирует tfjs-vis с помощью тега `<script>`:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs-vis@latest">
</script>
```

[1] Этот API построения диаграмм основывается на библиотеке визуализации Vega: https://vega.github.io/vega/.

Основная библиотека TensorFlow.js импортируется иначе:

```
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest">
</script>
```

То же относится к пакетам tfjs-vis и TensorFlow.js системы управления пакетами npm (`@tensorflow/tfjs-vis` и `@tensorflow/tfjs` соответственно). В веб-страницах или JavaScript-программах, зависящих как от TensorFlow.js, так и от tfjs-vis, необходимо импортировать обе зависимости.

Линейные диаграммы

Вероятно, чаще всего используемый тип диаграмм — *линейная диаграмма* (line chart) (ломаная, соединяющая точки данных). Она содержит горизонтальную и вертикальную оси, часто называемые *ось X* и *ось Y* соответственно. Подобные визуализации встречаются повсюду. Например, с помощью линейной диаграммы, по горизонтальной оси которой откладывается время суток, а по вертикальной — показания термометра, можно построить график изменения температуры за день. По горизонтальной оси линейной диаграммы можно откладывать не только время. Например, с помощью линейной диаграммы можно показать взаимосвязь между терапевтическим эффектом лекарства от давления (степень снижения давления) и его дозой (количеством принимаемого за день лекарства). Подобный график называется *кривой «доза — эффект»* (dose-response curve). Еще один хороший пример не связанной с временем линейной диаграммы — обсуждавшаяся в главе 3 кривая ROC. В ней ни ось *X*, ни ось *Y* никакого отношения ко времени не имеют (а представляют собой ложнопозитивные и истиннопозитивные результаты работы классификатора).

Для создания линейной диаграммы с помощью `tfvis.render` используется функция `linechart()`. Как демонстрирует первый пример в CodePen (и листинг 7.1), эта функция принимает три аргумента.

- Первый аргумент представляет собой HTML-элемент, в котором должна отрисовываться диаграмма. Достаточно пустого элемента `<div>`.
- Второй аргумент содержит значения точек данных на графике и представляет собой простой Java-объект в старом стиле (POJO), поле `value` которого указывает на массив, состоящий из определенного количества пар $x — y$, представляемых с помощью объектов POJO с полями `x` и `y`. Значения `x` и `y`, разумеется, отражают координаты x и y точек данных.
- Третий аргумент, необязательный, содержит дополнительные поля конфигурации для линейной диаграммы. В данном примере мы используем поле `width` для задания ширины итогового графика (в пикселах). В следующих примерах вы встретите и другие поля конфигурации[1].

[1] Полную документацию API tfjs-vis со списком прочих полей конфигурации этой функции можно найти по адресу https://js.tensorflow.org/api_vis/latest/.

Листинг 7.1. Создание простой линейной диаграммы с помощью tfvis.render.linechart()

```
let values = [{x: 1, y: 20}, {x: 2, y: 30},
              {x: 3, y: 5}, {x: 4, y: 12}];
tfvis.render.linechart(document.getElementById('plot1'),
                       {values},
                       {width: 400});
```

Ряд данных представляет собой массив пар x — y

Второй аргумент — это Object, содержащий ключ values

Первый аргумент представляет собой HTML-элемент, в котором будет отрисовываться диаграмма. В данном случае 'plot1' — это идентификатор пустого элемента

В третьем аргументе передаются пользовательские настройки. В данном случае они включают только ширину графика

Созданная с помощью кода из листинга 7.1 линейная диаграмма приведена слева на рис. 7.1. Это простая кривая, всего из четырех точек данных. Впрочем, функция `linechart()` поддерживает кривые, состоящие из намного большего числа точек данных (скажем, тысяч). Однако если пытаться построить график слишком большого количества точек данных сразу, рано или поздно можно столкнуться с ограниченностью ресурсов, доступных браузеру. Эти ограничения зависят от конкретных платформы и браузера, их приходится выяснять эмпирическим путем. Вообще говоря, рекомендуется ограничивать размер визуализируемых интерактивно данных ради плавной работы и быстроты отклика UI.

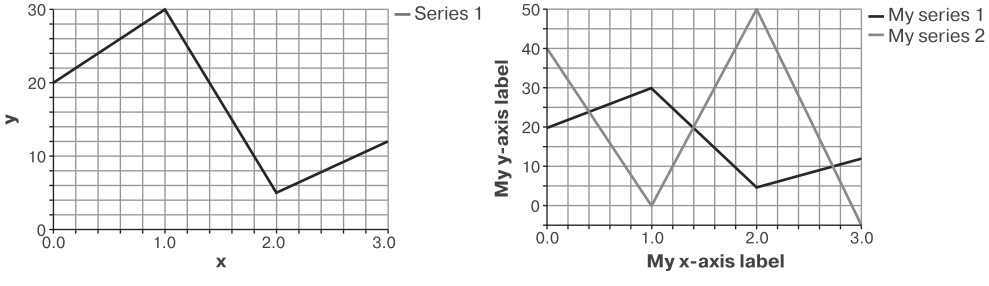


Рис. 7.1. Линейные диаграммы, созданные с помощью tfvis.render.linechart(). *Слева:* один ряд данных, создан с помощью кода из листинга 7.1. *Справа:* два ряда данных в той же системе координат, создан с помощью кода из листинга 7.2

Иногда возникает необходимость построить графики двух кривых на одной диаграмме, чтобы показать их взаимосвязь (например, чтобы сравнить их друг с другом). С помощью `tfvis.render.linechart()` можно создать и такие графики. Соответствующий пример приведен в правом блоке на рис. 7.1 и в коде из листинга 7.2.

Такие графики называются *диаграммами нескольких рядов данных* (multi-series chart), а отдельные кривые — *рядами данных*. Для создания подобной диаграммы необходимо включить в первый аргумент функции `linechart()` дополнительное поле, `series`, значение которого представляет собой массив строк — названий рядов данных, визуализируемых в виде легенды итогового графика. В примере кода мы назвали наши ряды данных `'My series 1'` и `'My series 2'`.

Листинг 7.2. Создание линейной диаграммы с двумя рядами данных с помощью функции tfvis.render.linechart()

```
values = [
  [{x: 1, y: 20}, {x: 2, y: 30}, {x: 3, y: 5}, {x: 4, y: 12}],
  [{x: 1, y: 40}, {x: 2, y: 0}, {x: 3, y: 50}, {x: 4, y: -5}]
];
let series = ['My series 1', 'My series 2'];
tfvis.render.linechart(
    document.getElementById('plot2'), {values, series}, {
  width: 400,
  xLabel: 'My x-axis label',
  yLabel: 'My y-axis label'
});
```

Для отображения в одной системе координат нескольких рядов данных формируем массив values из нескольких массивов пар x — y

При построении графиков нескольких рядов данных необходимо указывать названия рядов

Переопределяем используемые по умолчанию метки осей X и Y

Для диаграммы нескольких рядов данных необходимо также должным образом задать поле `value` первого аргумента. В нашем первом примере мы передали в него массив точек данных, в случае же диаграммы нескольких рядов данных нужно передать массив массивов. Каждый элемент вложенного массива представляет собой точки данных одного из рядов в том же формате, что и массив `values` в листинге 7.1, где мы строили диаграмму с одним рядом данных. Следовательно, длина вложенного массива должна соответствовать длине массива `series`, иначе возникнет ошибка.

Созданная с помощью листинга 7.2 диаграмма показана справа на рис. 7.1. Как вы видите на диаграмме, библиотека tfjs-vis выбрала два разных цвета (голубой и оранжевый) для визуализации двух кривых. Подобная цветовая схема, как правило, довольно удачна благодаря хорошей различимости этих цветов. При большем количестве рядов данных автоматически выбираются новые цвета.

Два ряда данных в этом примере диаграммы несколько необычны, поскольку их множества значений по координате x в точности совпадают (1, 2, 3 и 4). Впрочем, в общем случае значения по координате x различных рядов данных диаграммы с несколькими рядами не обязательно совпадают. Можете попробовать построить такую диаграмму в упражнении 1 в конце главы. Но учтите, что строить два подобных графика на одной диаграмме — не всегда удачная идея. Например, если диапазоны значений по оси Y двух кривых сильно различаются и вообще не пересекаются, будет непросто разобрать нюансы каждой из кривых при построении их графиков на одной линейной диаграмме. В подобных случаях лучше строить их на отдельных линейных диаграммах.

Стоит отметить в листинге 7.2 и пользовательские метки осей координат. Чтобы пометить оси X и Y выбранными строковыми метками, мы воспользовались полями `xLabel` и `yLabel` объекта конфигурации (третий из передаваемых в функцию `linechart()` аргументов). В общем случае всегда следует задавать метки для осей координат, чтобы сделать диаграммы понятнее. Если не задать поля `xLabel` и `yLabel`, tfjs-vis укажет для осей метки `x` и `y`, как это произошло в листинге 7.1 и в левой части рис. 7.1.

Диаграммы рассеяния

Еще одна разновидность диаграмм, которую можно построить с помощью `tfvis.render`, — *диаграммы рассеяния* (scatter plots), они же *точечные диаграммы*. Наиболее заметное отличие диаграмм рассеяния от линейных диаграмм — отсутствие на первых отрезков, соединяющих точки данных, благодаря чему они подходят для случаев, когда порядок точек данных неважен. Например, с помощью диаграммы рассеяния можно построить график населения нескольких стран относительно их ВВП на душу населения. На подобном графике главное — отношения *x*- и *y*-значений, а не упорядоченность точек данных.

В пространстве имен `tfvis.render` для создания диаграмм рассеяния служит функция `scatterplot()`. Как демонстрирует пример в листинге 7.3, функция `scatterplot()` способна визуализировать сразу несколько рядов данных, аналогично `linechart()`. На самом деле API `scatterplot()` и `linechart()` практически идентичны, как видно из сравнения листинга 7.2 с листингом 7.3. Созданная с помощью листинга 7.3 диаграмма рассеяния показана на рис. 7.2.

Листинг 7.3. Создание диаграммы рассеяния с помощью функции tfvis.render.scatterplot()

```
values = [
  [{x: 20, y: 40}, {x: 32, y: 0}, {x: 5, y: 52}, {x: 12, y: -6}],
  [{x: 15, y: 35}, {x: 0, y: 9}, {x: 7, y: 28}, {x: 16, y: 8}]
];
series = ['My scatter series 1', 'My scatter series 2'];
tfvis.render.scatterplot(
    document.getElementById('plot4'),
  {values, series},
   {
    width: 400,
    xLabel: 'My x-values',
    yLabel: 'My y-values'
  });
```

Как и в linechart(), для отображения нескольких рядов данных на одной диаграмме рассеяния используется массив массивов пар x — y

Не забывайте задавать метки осей координат

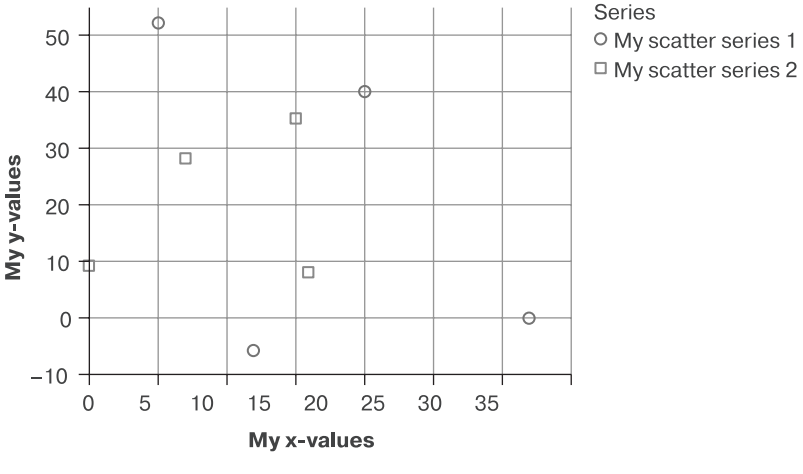


Рис. 7.2. Диаграмма рассеяния с двумя рядами данных. Создана с помощью кода из листинга 7.3

Столбчатые диаграммы

Как ясно из их названия, в *столбчатой диаграмме* (bar chart) значения величин отображаются с помощью прямоугольных зон (столбцов). Эти столбцы обычно начинаются с нуля внизу, чтобы по относительной высоте столбца можно было судить о соотношениях величин. Поэтому столбчатые диаграммы удобны, когда важны именно соотношения между величинами. Например, такие диаграммы часто используются для отображения годовых доходов компании за несколько лет. В подобном случае из относительных высот столбцов интуитивно ясно, как меняется доход по кварталам. Это отличает столбчатые диаграммы от линейных диаграмм и графиков рассеяния, в которых значения далеко не всегда привязаны к нулю.

Для создания столбчатой диаграммы с помощью `tfvis.render` используется функция `barchart()`. Пример приведен в листинге 7.4. Созданная с помощью этого кода столбчатая диаграмма показана на рис. 7.3. API функции `barchart()` аналогичен API функций `linechart()` и `scatterplot()`. Впрочем, следует отметить важное их отличие. Первый передаваемый в `barchart()` аргумент представляет собой не состоящий из поля `value` объект, а простой массив пар «индекс — значение». Горизонтальные значения задаются с помощью поля, называющегося не `x`, а `index`. Аналогично вертикальные значения задаются с помощью поля, называющегося не `y`, а `value`. Почему так? Дело в том, что горизонтальные значения столбца в столбчатой диаграмме не обязательно числа. Они могут быть как строковыми значениями, так и числами, как демонстрирует наш пример на рис. 7.3.

Листинг 7.4. Создание столбчатой диаграммы с помощью функции tfvis.render.barchart()

```
const data = [
    {index: 'foo', value: 1},{index: 'bar', value: 7},
    {index: 3, value: 3},
    {index: 5, value: 6}];
  tfvis.render.barchart(document.getElementById('plot5'), data, {
    yLabel: 'My value',
    width: 400
  });
```

Обратите внимание, что индекс столбчатой диаграммы может быть числовым или строковым. Учтите также, что порядок элементов важен

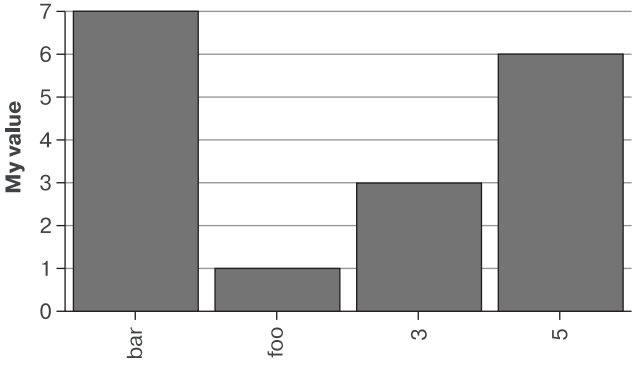


Рис. 7.3. Столбчатая диаграмма, состоящая как из строковых, так и из числовых столбцов. Создана с помощью кода из листинга 7.4

Гистограммы

Три вышеописанных типа графиков позволяют строить зависимости определенных величин. Иногда не столь важны подробные количественные значения, как *распределение* значений. Представьте себе экономиста, изучающего данные по годовым доходам семей из результатов общегосударственной переписи. Для него наибольший интерес представляют не подробные показатели доходов. В них содержится слишком много информации (да, иногда слишком много информации тоже плохо!). Вместо них он предпочел бы более краткие сводные показатели дохода. Ему интересно распределение этих значений, то есть сколько из них меньше $20 000, сколько в диапазоне от $20 000 до 40 000 или от $40 000 до 60 000 и т. д. Для решения подобной задачи визуализации и служат *гистограммы* (histograms).

В гистограмме значения распределяются по *интервалам* (bins), иногда называемым *«корзинами»*. Эти интервалы — просто непрерывные диапазоны значений величины со своими нижними и верхними границами. Обычно выбираются смежные интервалы, чтобы охватить все возможные значения. В предыдущем примере экономист мог бы воспользоваться следующими интервалами значений: 0 ~ 20 тыс., 20 тысяч ~ 40 тысяч, 40 тысяч ~ 60 тысяч и т. д. После выбора множества из `N` интервалов значений пишется программа для подсчета количества отдельных точек данных, попадающих в каждый из интервалов, в результате выполнения которой получается *N* чисел (по одному для каждого интервала значений). А если построить график этих чисел с помощью вертикальных столбцов, то и получится гистограмма.

Все вышеописанные шаги реализованы в функции `tfvis.render.histogram()`. Она освобождает вас от забот по определению границ интервалов и подсчету числа примеров данных в них. Для вызова `histogram()` достаточно передать в нее массив чисел, как показано в листинге 7.5. Никакой упорядоченности этих чисел не требуется.

Листинг 7.5. Визуализация распределения значений с помощью tfvis.render.histogram()

```
const data = [1, 5, 5, 5, 5, 10, -3, -3];
tfvis.render.histogram(document.getElementById('plot6'), data, {
  width: 400
});

// Гистограмма с задаваемым пользователем
// числом интервалов значений.
// Данные — такие же, как и выше.
tfvis.render.histogram(document.getElementById('plot7'), data, {
  maxBins: 3,
  width: 400
});
```

«Корзины» генерируются автоматически

Задаем количество «корзин» явным образом

В листинге 7.5 есть два несколько отличающихся вызова функции `histogram()`. При первом вызове не задается никаких пользовательских опций, за исключением ширины графика, и `histogram()` рассчитывает интервалы значений с помощью встроенных эвристических механизмов. В итоге получается семь «корзин»: –4 ~ –2, –2 ~ 0, 0 ~ 2... 8 ~ 10, как показано слева на рис. 7.4. При разбиении значений по этим интервалам самое большое значение (4) оказывается в «корзине» 4 ~ 6, поскольку

четыре из значений массива данных равны 5. Три «корзины» гистограммы (–2 ~ 0, 2 ~ 4 и 6 ~ 8) вообще пусты, поскольку ни один из элементов точек данных ни в одну из них не попадает.

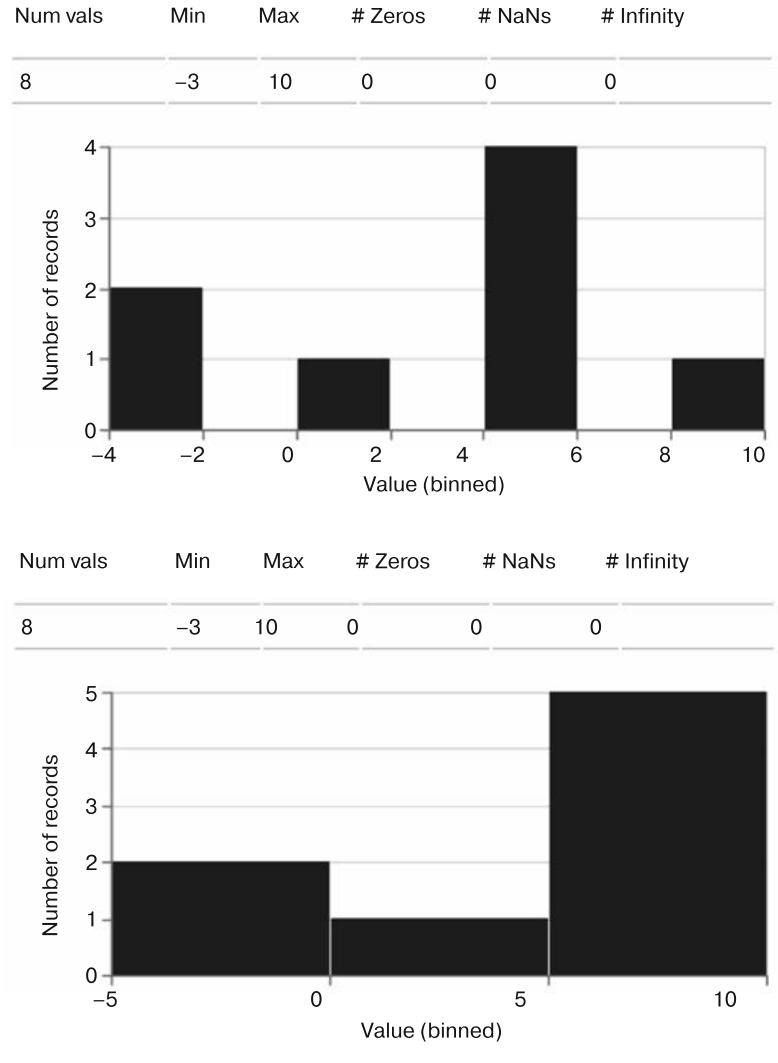


Рис. 7.4. Гистограммы одних и тех же данных с автоматически вычисленным количеством интервалов значений *(слева)* и количеством, заданным явным образом *(справа)*. Код, генерирующий эти гистограммы, приведен в листинге 7.5

Возникает вопрос: а не слишком ли много интервалов значений для наших конкретных точек данных было создано по умолчанию? При меньшем числе интервалов значений вероятность пустых интервалов будет меньше. Для переопределения используемого по умолчанию эвристического алгоритма можно воспользоваться полем

конфигурации `maxBins` и ограничить количество интервалов значений. Именно это мы делаем во втором вызове `histogram()` в листинге 7.5, результат которого приведен справа на рис. 7.4. Как видите, при ограничении числа интервалов значений до трех все «корзины» оказываются непустыми.

Карты интенсивности

Карта интенсивности (heatmap) отображает двумерный массив чисел в виде сетки раскрашенных ячеек, цвета которых отражают относительную величину элементов массива. Обычно для более низких значений используются более холодные цвета, например синий и зеленый, а для более высоких — более теплые, например оранжевый и красный. Поэтому такие графики и называются картами интенсивности[1]. Вероятно, наиболее часто встречающийся пример карт интенсивности в глубоком обучении — матрицы различий (см. пример iris-flower в главе 3) и матрицы внимания (см. пример date-conversion в главе 9). Для этой разновидности визуализации tfjs-vis предоставляет функцию `tfvis.render.heatmap()`.

Листинг 7.6 демонстрирует создание карты интенсивности для визуализации вымышленной матрицы различий, включающей три класса. Значение матрицы различий задается в поле `values` второго из входных аргументов. Названия классов — метки столбцов и строк карты интенсивности, задаются в массивах `xTickLabels` и `yTickLabels`. Не путайте эти промежуточные метки с `xLabel` и `yLabel` из третьего аргумента — метками осей координат *X* и *Y* в целом. Полученная в результате карта интенсивности приведена на рис. 7.5.

Листинг 7.6. Визуализация двумерных тензоров с помощью функции tfvis.render.heatmap()

```
tfvis.render.heatmap(document.getElementById('plot8'), {
  values: [[1, 0, 0], [0, 0.3, 0.7], [0, 0.7, 0.3]],
  xTickLabels: ['Apple', 'Orange', 'Tangerine'],
  yTickLabels: ['Apple', 'Orange', 'Tangerine']
}, {
  width: 500,
  height: 300,
  xLabel: 'Actual Fruit',
  yLabel: 'Recognized Fruit',
  colorMap: 'blues'
});
```

Передаваемый в функцию values может представлять собой вложенный JavaScript-массив (как здесь) или двумерный тензор типа tf.Tensor

Массив xTickLabels содержит метки для отдельных столбцов по оси X. Не путайте его с xLabel. Аналогично массив yTickLabels содержит метки для отдельных столбцов по оси Y

xLabel и yLabel содержат метки осей координат X и Y в целом, в отличие от xTickLabels и yTickLabels

Помимо используемой здесь карты цветов 'blues', существуют также 'greyscale' и 'viridian'

На этом наш краткий обзор четырех основных поддерживаемых `tfvis.render` типов диаграмм завершается. Если в будущем вы планируете заниматься визуализацией данных с помощью tfjs-vis, весьма вероятно, что вы часто будете их использовать. Таблица 7.1 подытоживает эти типы диаграмм, чтобы упростить вам выбор подходящей для конкретной задачи визуализации.

[1] В дословном переводе с английского — «тепловая карта». — *Примеч. пер.*

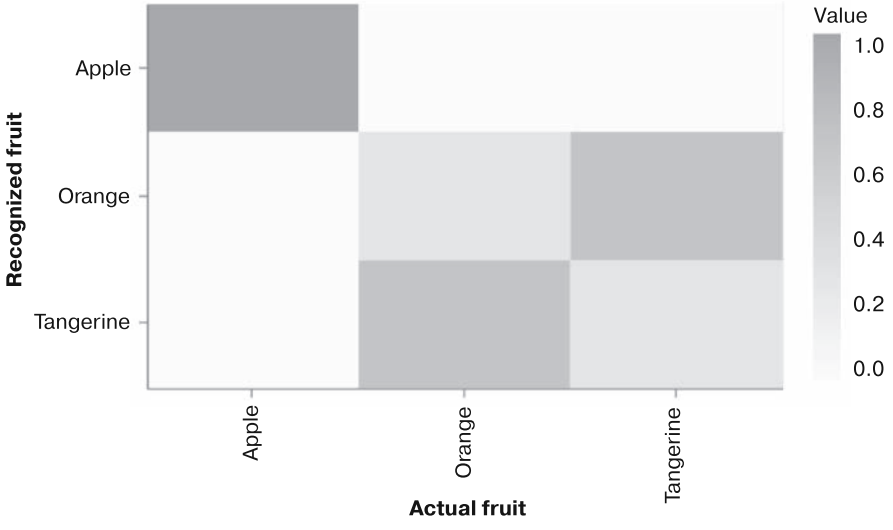


Рис. 7.5. Карта интенсивности, визуализируемая кодом из листинга 7.6. Она отображает фиктивную матрицу различий, включающую три класса

Таблица 7.1. Резюме пяти основных типов диаграмм, поддерживаемых tfjs-vis в пространстве имен tfvis.render

Название диаграммы	**Соответствующие функции tfjs-vis**	**Подходящие для их использования задачи визуализации и примеры из сферы машинного обучения**
Линейная диаграмма	`tfvis.render.linechart()`	Зависимость одного скалярного значения (y) от другого (x) с внутренней упорядоченностью (время, доза и т. д.). В одной системе координат можно построить графики нескольких рядов данных: например, метрик для обучающего и проверочного набора данных, каждый — в зависимости от номера эпохи обучения
Диаграмма рассеяния	`tfvis.render.scatterplot()`	Пары скалярных значений x — y без внутренней упорядоченности, например отношения между собой двух числовых столбцов CSV набора данных. В одной системе координат можно построить графики нескольких рядов данных
Столбчатая диаграмма	`tfvis.render.barchart()`	Набор значений, относящихся к небольшому числу категорий, например показатели безошибочности (в процентах) для нескольких моделей на одной задаче классификации
Гистограмма	`tfvis.render.histogram()`	Набор значений, в котором нас интересует главным образом распределение, например распределение значений параметров ядра плотного слоя

Продолжение ⇩

Таблица 7.1 *(продолжение)*

Название диаграммы	Соответствующие функции tfjs-vis	Подходящие для их использования задачи визуализации и примеры из сферы машинного обучения
Карта интенсивности	`tfvis.render.heathmap()`	Двумерный массив чисел для визуализации в виде двумерной сетки ячеек с цветовым кодированием всех элементов для отражения величины соответствующих значений, например матрица различий многоклассового классификатора (см. раздел 3.3) или матрица внимания модели преобразования последовательностей в последовательности (см. раздел 9.3)

7.1.2. Комплексный практический пример: визуализация метеорологических данных с помощью tfjs-vis

В примерах на CodePen в предыдущем разделе использовались небольшие по объему зашитые в код данные. В этом разделе мы покажем, как воспользоваться возможностями tfjs-vis для гораздо большего и интересного набора данных из практики. Мы продемонстрируем подлинную мощь его API и поясним, в чем ценность подобных визуализаций данных в браузере. Этот пример также проиллюстрирует некоторые нюансы и хитрости, с которыми можно столкнуться при использовании API создания диаграмм на практике.

В этом примере воспользуемся набором данных Jena-weather-archive. Он включает измерения, собранные в городе Йена, Германия, со множества метеорологических инструментов за восемь лет (с 2009-го по 2017-й). Этот набор данных, который можно скачать со страницы Kaggle (см. http://www.kaggle.com/pankrzysiu/weather-archive-jena), представляет собой CSV-файл размером 42 Мбайт, состоящий из 15 столбцов. Первый столбец — метка даты/времени, а остальные представляют собой различные метеорологические данные, например температуру (`T deg(C)`), атмосферное давление (`p (mbar)`), относительную влажность (`rh (%s)`), скорость ветра (`wv (m/s)`) и т. д. Если внимательнее взглянуть на метки даты/времени, можно заметить, что их разделяют промежутки времени десять минут, поскольку измерения производились каждые десять минут. Это очень многообещающий набор данных для визуализации, исследования и применения машинного обучения. В следующих разделах мы будем пытаться сформировать на его основе прогнозы погоды с помощью различных моделей машинного обучения. В частности, будем предсказывать температуру в конкретный день на основе метеорологических данных за предыдущие десять дней. Но прежде, чем заняться этой увлекательной задачей прогноза погоды, последуем принципу «Всегда изучай имеющиеся данные, прежде чем применять к ним модели машинного обучения» и разберемся, как визуализировать эти данные с помощью tfjs-vis интуитивно понятным образом.

Скачать и запустить пример Jena-weather можно с помощью следующих команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/jena-weather
yarn
yarn watch
```

Эффективная визуализация за счет ограничения объема данных

Набор данных Jena-weather довольно велик. При размере файла 42 Мбайт он превышает все предыдущие CSV и табличные наборы данных в книге. В итоге возникает две проблемы.

- Первая — для компьютера: при построении графика данных за все восемь лет сразу ресурсов вкладки браузера не хватит, она перестанет реагировать и, вероятно, произойдет фатальный сбой. Даже если ограничиться одним столбцом из 14, все равно необходимо отобразить примерно 420 000 точек данных, что превышает возможности визуализации tfjs-vis (или любой другой современной JavaScript-библиотеки построения графиков, если уж на то пошло).
- Вторая — для пользователя: не так уж просто понять, что к чему, при таком большом объеме данных. Например, как увидеть и извлечь полезную информацию из всех 420 000 точек данных сразу? Как и у компьютера, производительность человеческого мозга в отношении обработки данных ограниченна. Задача создателя визуализации как раз и состоит в том, чтобы эффективно представить наиболее релевантные и несущие максимум информации аспекты данных.

  Для решения этих проблем воспользуемся тремя уловками.

- Вместо того чтобы строить график данных за все восемь лет сразу, предоставим пользователю возможность выбора промежутка времени с помощью интерактивного UI. Для этого предназначен раскрывающийся список Time span в UI (рис. 7.6 и 7.7). Варианты временных интервалов включают Day, Week, 10 Days, Month, Year и Full. Последний вариант соответствует полному промежутку времени в восемь лет. В случае всех прочих промежутков времени у пользователя есть возможность передвигаться по времени вперед и назад, нажимая кнопки, на которых изображены стрелка влево и стрелка вправо.
- Для любого промежутка времени, превышающего неделю, производится *понижающая дискретизация* временного ряда перед построением его графика на экране. Например, рассмотрим промежуток времени Month (30 дней). Полный массив данных для этого промежутка содержит примерно $30 \times 24 \times 6 = 4320$ точек данных. Как видно из кода в листинге 7.7, мы выводим на график только каждую шестую точку данных при отображении данных за месяц, что сокращает число точек данных до 720, — это существенная экономия необходимых для визуализации ресурсов. Но для невооруженного глаза такое шестикратное снижение числа точек данных остается практически незаметным.
- Аналогично раскрывающемуся списку Time span мы включаем в UI список, позволяющий пользователю выбрать вид метеорологических данных для построения

графика. Обратите внимание на раскрывающиеся меню Data series 1 и Data series 2. С их помощью пользователь может строить линейные диаграммы любого одного или двух из 14 столбцов в тех же осях координат.

В листинге 7.7 показан код построения графиков, приведенных на рис. 7.6. И хотя в коде вызывается `tfvis.render.linechart()`, аналогично примеру CodePen в предыдущем подразделе, он намного абстрактнее кода из предыдущих листингов. Дело в том, что на этой веб-странице нам нужно отложить выбор того, графики каких величин необходимо построить, чтобы провести его в соответствии с состоянием UI.

Листинг 7.7. Метеорологические данные в виде линейной диаграммы с несколькими рядами данных (из jena-weather/index.js)

```
function makeTimeSerieChart(
    series1, series2, timeSpan, normalize, chartContainer) {
  const values = [];
  const series = [];
  const includeTime = true;
  if (series1 !== 'None') {
    values.push(jenaWeatherData.getColumnData(
        series1, includeTime, normalize, currBeginIndex,
        TIME_SPAN_RANGE_MAP[timeSpan],
        TIME_SPAN_STRIDE_MAP[timeSpan]));
    series.push(normalize ? `${series1} (normalized)` : series1);
  }
  if (series2 !== 'None') {
    values.push(jenaWeatherData.getColumnData(
        series2, includeTime, normalize, currBeginIndex,
        TIME_SPAN_RANGE_MAP[timeSpan],
        TIME_SPAN_STRIDE_MAP[timeSpan]));
    series.push(normalize ? `${series2}
                (normalized)` : series2);
  }
  tfvis.render.linechart({values, series: series}, chartContainer, {
    width: chartContainer.offsetWidth * 0.95,
    height: chartContainer.offsetWidth * 0.3,
    xLabel: 'Time',
    yLabel: series.length === 1 ? series[0] : ''
  });
}
```

jenaWeatherData — вспомогательный объект для упорядочения и извлечения метеорологических данных из CSV-файла. См. jena-weather/data.js

Задаем промежуток времени для визуализации

Выбираем подходящий шаг свертки (коэффициент понижающей дискретизации)

Воспользуемся тем, что линейные диаграммы библиотеки tfjs-vis поддерживают построение нескольких рядов данных на одном графике

Не забывайте задавать метки для осей координат

Поэкспериментируйте с UI визуализации данных, в котором вы найдете множество интересных погодных закономерностей. Например, в верхнем блоке рис. 7.6 показаны колебания нормализованной температуры (`T deg(C)`) и нормализованного атмосферного давления (`p (mbar)`) за период десять дней. На кривой температуры ясно различим суточный цикл: температура достигает максимума около полудня, а минимума — вскоре после полуночи. Помимо суточного цикла, на этом рисунке можно видеть и более глобальную тенденцию (постепенный рост температуры) за десять дней. И напротив, на кривой атмосферного давления никакой четкой закономерности не заметно. В нижнем блоке того же рисунка приведены те же измерения

за год. Здесь виден годичный цикл температуры: максимум в августе и минимум в январе. Атмосферное давление опять же демонстрирует намного менее четкую закономерность, чем температура, при такой временной шкале.

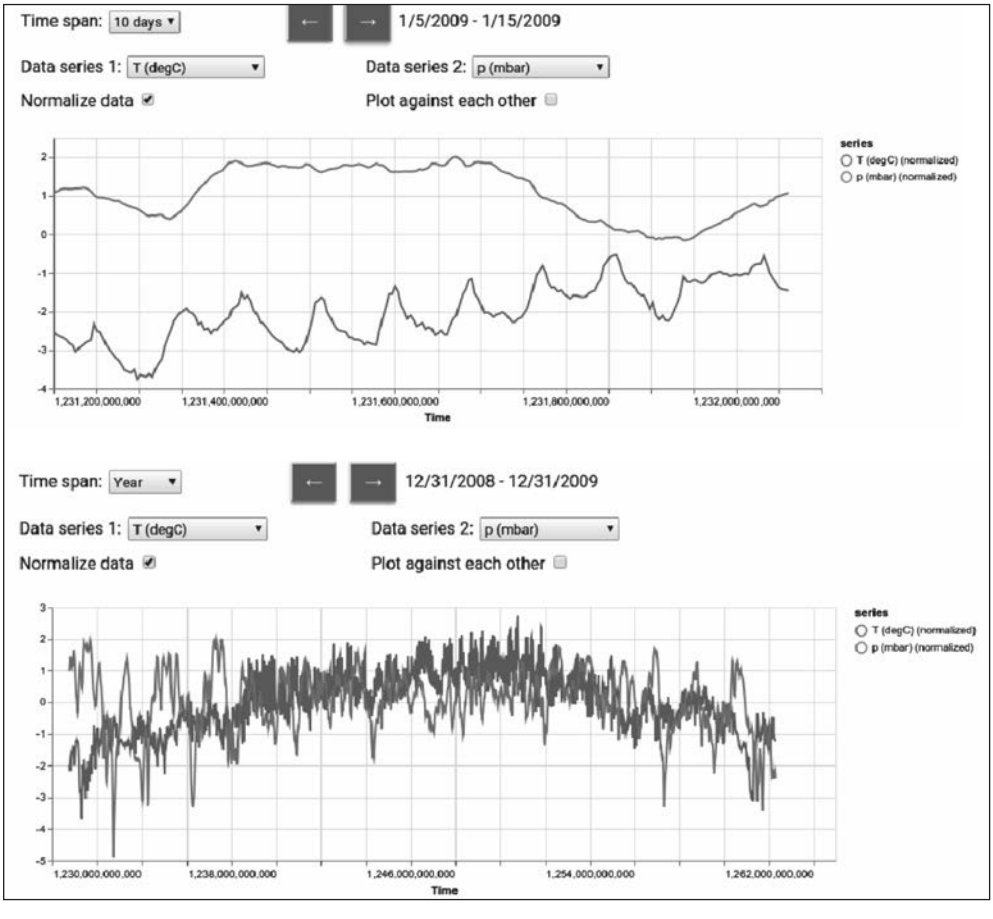


Рис. 7.6. Линейные диаграммы температуры (T deg(C)) и атмосферного давления (p (mbar)) из набора данных Jena-weather-archive при двух различных временных шкалах. *Сверху:* промежуток времени десять дней. Обратите внимание на суточный цикл на кривой температуры. *Снизу:* промежуток времени один год. Обратите внимание на годичный цикл на кривой температуры и некоторую стабилизацию атмосферного давления весной и летом, по сравнению с другими временами года

Давление в течение года может меняться довольно беспорядочно, хотя, похоже, оно варьируется летом несколько меньше, чем зимой. Глядя на одни и те же измерения в различных временных шкалах, можно заметить множество интересных закономерностей, которые практически невозможно увидеть по исходным данным в числовом формате CSV.

Возможно, вы обратили внимание, что в диаграммах на рис. 7.6 отображаются нормализованные, а не абсолютные значения температуры и атмосферного давления, поскольку при создании этих графиков был выбран флажок Normalize data. Мы вкратце уже говорили про нормализацию, когда обсуждали модель Boston-housing в главе 2. Там нормализация состояла из вычитания среднего значения и деления полученного результата на среднеквадратичное отклонение, а делалось это для того, чтобы повысить результативность обучения модели. Здесь нормализация состоит точно в том же, однако служит не только для повышения безошибочности модели машинного обучения (что мы обсудим в следующем разделе), но и для визуализации. Почему так? Если вы снимете флажок Normalize data при отображении на диаграмме температуры и атмосферного давления, то сразу же поймете почему. Диапазон измерений температуры — от –10 до 40 градусов (по шкале Цельсия), а атмосферного давления — от 980 до 1000. Построение в одной системе координат без нормализации двух величин с такими различиями диапазонов значений приводит к расширению оси *Y* до очень большого диапазона, в результате чего обе кривые выглядят практически как прямые линии. Нормализация позволяет избежать этой проблемы посредством приведения всех измерений к распределению с нулевым математическим ожиданием и единичным среднеквадратичным отклонением.

На рис. 7.7 приведен пример построения графика зависимости двух измерений метеорологических показателей друг от друга в виде диаграммы рассеяния — режим, который можно активизировать с помощью кнопки-флажка Plot against each other, необходимо только убедиться, что ни в одном из списков Data series не выбрано None. Код создания подобных диаграмм рассеяния аналогичен функции `makeTimeSerieChart()` в листинге 7.7, а потому мы его опустим ради краткости. Можете изучить его в том же файле (`jena-weather/index.js`), если вам интересны подробности.

Этот пример диаграммы рассеяния показывает зависимость между нормализованной плотностью воздуха (по оси *Y*) и нормализованной температурой (по оси *X*). Сразу заметна довольно сильная корреляция между этими двумя величинами: плотность воздуха падает с ростом температуры. В этом примере промежуток времени равен десяти дням, но вы можете убедиться, что эта тенденция более или менее сохраняется и при других интервалах времени. Визуализация подобной корреляции величин на диаграммах рассеяния не доставляет трудностей, но по данным в текстовом формате обнаружить ее намного сложнее — еще один пример пользы визуализации данных.

7.2. Визуализация моделей после обучения

В предыдущих разделах мы показали, чем может быть полезна визуализация данных. В этом разделе рассмотрим, как визуализировать различные аспекты моделей после их обучения, чтобы почерпнуть из них полезную информацию. Для этого сосредоточим внимание преимущественно на сверточных сетях с изображениями в качестве входных сигналов, поскольку они широко распространены и дают интересные результаты визуализации.

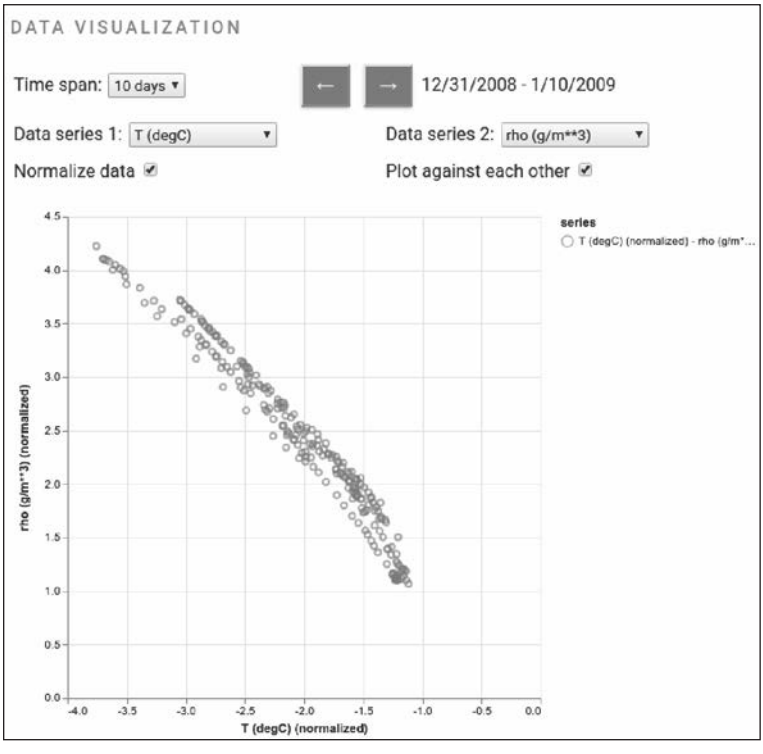


Рис. 7.7. Пример диаграммы рассеяния из демонстрации Jena-weather. График отражает зависимость между плотностью воздуха (rho, вертикальная ось координат) и температурой (T, горизонтальная ось координат) за десять дней с заметной отрицательной корреляцией

Наверное, вы слышали, как глубокие нейронные сети называют «черными ящиками». Но не стоит торопиться считать сложным извлечение информации изнутри нейронной сети во время обучения или выполнения вывода. Напротив, заглянуть внутрь происходящего в каждом из слоев написанной на TensorFlow.js модели совсем несложно[1]. Более того, если говорить о сверточных сетях, то усваиваемые ими внутренние представления чрезвычайно удобны для визуализации в основном потому, что отражают визуальные концепты. С 2013 года разработано

[1] На самом деле это высказывание говорит только о сложности описания в простых словах происходящего в глубоких нейронных сетях большого количества математических операций, по сравнению с некоторыми другими типами алгоритмов машинного обучения, например деревьями принятия решений и логистической регрессией. В частности, в случае дерева принятия решений можно обходить точки ветвления по очереди и объяснять причину выбора конкретной ветки простой фразой наподобие «поскольку коэффициент X превышает 0,35». Эта задача не относится к теме данного раздела и называется интерпретируемостью модели (model interpretability).

множество подходов визуализации и интерпретации этих представлений. Поскольку рассматривать здесь все их не имеет смысла, мы охватим лишь три основных и наиболее полезных.

- *Визуализация выходных сигналов промежуточных слоев (промежуточных функций активации) сверточной сети*. Этот подход удобен для выяснения того, как последовательные слои сверточной сети преобразуют свои входные сигналы, и для получения первого представления о визуальных признаках, усвоенных отдельными фильтрами сверточной сети.
- *Визуализация фильтров сверточной сети путем поиска наиболее активирующих их входных изображений*. Подход удобен для выяснения того, к каким визуальным паттернам или концептам чувствителен каждый из фильтров.
- *Визуализация карт интенсивности активаций классов во входном изображении*. С помощью этой методики можно понять, какие части входного изображения играют наиболее важную роль в генерации сверточной сетью итогового результата классификации, благодаря чему можно также выяснить, как сверточная сеть получает выходной сигнал, и произвести «отладку» неправильных выходных результатов.

Код демонстрации этих методик вы можете найти в примере visualize-convnet из репозитория tfjs-examples. Для его запуска выполните следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/visualize-convnet
yarn && yarn visualize
```

Команда `yarn visualize` отличается от уже знакомой вам по предыдущим примерам команды `yarn watch`. Помимо сборки и запуска веб-страницы, она выполняет еще некоторые действия вне браузера. Во-первых, устанавливает необходимые библиотеки Python, а затем скачивает модель VGG16 (широко известная и часто используемая глубокая сверточная сеть) и преобразует ее в формат TensorFlow.js. Модель VGG16 предобучена на масштабном наборе данных ImageNet и доступна в виде приложения Keras. По завершении преобразования модели `yarn visualize` выполняет в tfjs-node набор исследований преобразованной модели. Почему в tfjs-node, а не в браузере? Потому что VGG16 — относительно большая сверточная сеть[1]. В результате некоторые из этих шагов требуют значительных вычислительных ресурсов и выполняются намного быстрее в менее стесненной в смысле ресурсов среде Node.js. Еще больше ускорить эти вычисления можно, воспользовавшись модулем tfjs-node-gpu вместо применяемого по умолчанию tfjs-node (для этого вам потребуется GPU с поддержкой CUDA с установленными нужными драйверами и библиотеками; см. приложение А):

```
yarn visualize --gpu
```

[1] Чтобы ощутить размеры VGG16, просто задумайтесь — общий размер ее весовых коэффициентов составляет более 528 Мбайт, по сравнению с размером весовых коэффициентов MobileNet, составляющих менее 10 Мбайт.

По завершении вычислений в Node.js, требующих большого объема шагов, генерируется набор файлов изображений в каталоге `dist/`. Наконец, команда `yarn visualize` компилирует и запускает веб-сервер для набора статических веб-файлов, включая эти изображения, а также открывает страницу `index` в браузере.

У команды `yarn visualize` есть несколько флагов настроек. Например, по умолчанию она выполняет вычисления и визуализацию по восьми фильтрам для каждого интересующего нас сверточного слоя. Это количество фильтров можно изменить с помощью флага `--filters`: например, `yarn visualize --filters 32`. Кроме того, по умолчанию команда `yarn visualize` использует входное изображение `cat.jpg`, поставляемое вместе с исходным кодом. Но можно указать и другие файлы изображений благодаря флагу `--image`[1]. А теперь взглянем на результаты визуализации для изображения `cat.jpg` и 32 фильтров.

7.2.1. Визуализация внутренних функций активации сверточной сети

В этом подразделе мы займемся расчетом и отображением карт признаков, сгенерированных различными сверточными слоями модели VGG16 на основе заданного входного изображения. Эти карты признаков называют *внутренними* активациями, поскольку они представляют собой не итоговый выходной сигнал модели (вектор длиной 1000, отражающий оценки вероятностей для 1000 классов ImageNet), а промежуточные шаги вычислений модели. Эти внутренние активации демонстрируют, как входной сигнал разбивается на различные усваиваемые моделью признаки.

Как вы помните из главы 4, форма выходного сигнала сверточного слоя — `[numExamples, height, width, channels]`. В данном случае речь идет об одном входном изображении, так `numExamples` = 1. Нам нужно визуализировать выходные сигналы каждого из сверточных слоев по трем оставшимся измерениям: высота, ширина и каналы. Высоту и ширину выходного сигнала сверточного слоя определяют размер его фильтра, дополнение нулями и шаг свертки, а также высота и ширина входного сигнала этого слоя. В общем случае они уменьшаются по мере продвижения в глубь сверточной сети. С другой стороны, число каналов при этом растет, поскольку сверточная сеть выделяет все больше и больше признаков через последовательные слои преобразований представления. Эти каналы сверточных слоев нельзя интерпретировать как различные компоненты цвета, они представляют собой измерения усвоенных признаков. Именно поэтому в нашей визуализации мы разбиваем их на отдельные блоки и отрисовываем в оттенках серого. На рис. 7.8 приведены активации пяти сверточных слоев модели VGG16 при входном изображении `cat.jpg`.

Самая заметная особенность внутренних активаций — по мере углубления в сеть они все больше отличаются от исходного входного сигнала. Первые слои (например, `block1_conv1`), похоже, кодируют относительно простые визуальные признаки,

[1] Поддерживаются наиболее распространенные форматы изображений, включая JPEG и PNG.

скажем границы фрагментов и цвета. Например, стрелка с меткой А указывает на внутреннюю активацию, судя по всему реагирующую на желтый и розовый цвета. Стрелка с меткой Б указывает на внутреннюю активацию, которая, очевидно, отвечает за границы фрагментов в определенных направлениях во входном изображении.

Но дальнейшие слои (например, `block4_conv2` и `block5_conv3`) демонстрируют паттерны активаций, все более и более отличающиеся от простых признаков на уровне отдельных пикселов во входном изображении. Например, стрелка с меткой В на рис. 7.8 указывает на фильтр в `block4_conv2`, по-видимому кодирующий черты мордочки кошки, включая уши, глаза и нос. Это конкретный пример выделения признаков по нарастающей, схематически показанного на рис. 4.6. Впрочем, заметим, что не все фильтры более поздних слоев сети можно настолько просто описать словами. Еще одно интересное наблюдение: «разреженность» карт активации также растет по мере продвижения в глубь сети. В первом слое на рис. 7.8 все фильтры активизированы (пиксельный паттерн не константный) входным изображением; в последнем же слое некоторые фильтры работают вхолостую (константный пиксельный паттерн; например, см. последнюю строку правого блока на рис. 7.8). Это значит, что признаки, кодируемые этими работающими вхолостую фильтрами, отсутствуют в данном конкретном входном изображении.

Вы только что стали свидетелем важной универсальной характеристики представлений, усваиваемых глубокими сверточными сетями: абстрактность выделяемых слоем признаков растет с глубиной слоя. Активации более глубоко расположенных слоев несут все меньше информации о нюансах входных данных и все больше и больше информации о целевом признаке (в данном случае о том, к какому из 1000 классов ImageNet принадлежит данное изображение). Таким образом, глубокая нейронная сеть фактически играет роль своего рода *конвейера выжимки информации*, на вход которого подаются необработанные данные, постепенно преобразуемые так, что аспекты входного сигнала, не относящиеся к решаемой задаче, отфильтровываются, а полезные для решения этой задачи — усиливаются и уточняются. И хотя мы демонстрировали это на примере сверточной сети, такая характеристика справедлива и для прочих типов глубоких нейронных сетей (например, многослойных перцептронов).

Те аспекты входных изображений, которые сверточная сеть считает полезными для решения поставленной задачи, могут отличаться от таковых для зрительной системы человека. Обучение сверточной сети основывается на данных, а потому чувствительно к систематическим ошибкам в обучающих примерах. Например, статья Марко Рибейро и его сотрудников, упомянутая в разделе «Материалы для дальнейшего чтения и изучения» в конце главы, описывает случай, когда изображение собаки ошибочно классифицируется как изображение волка из-за наличия в фоне изображения снега. Вероятно, потому, что обучающие изображения включали на фоне снега только волков, но не собак.

Из визуализации паттернов внутренних активаций глубокой сверточной сети мы почерпнули немало полезной информации. В следующих подразделах расскажем, как написать код на TensorFlow.js для извлечения этих внутренних активаций.

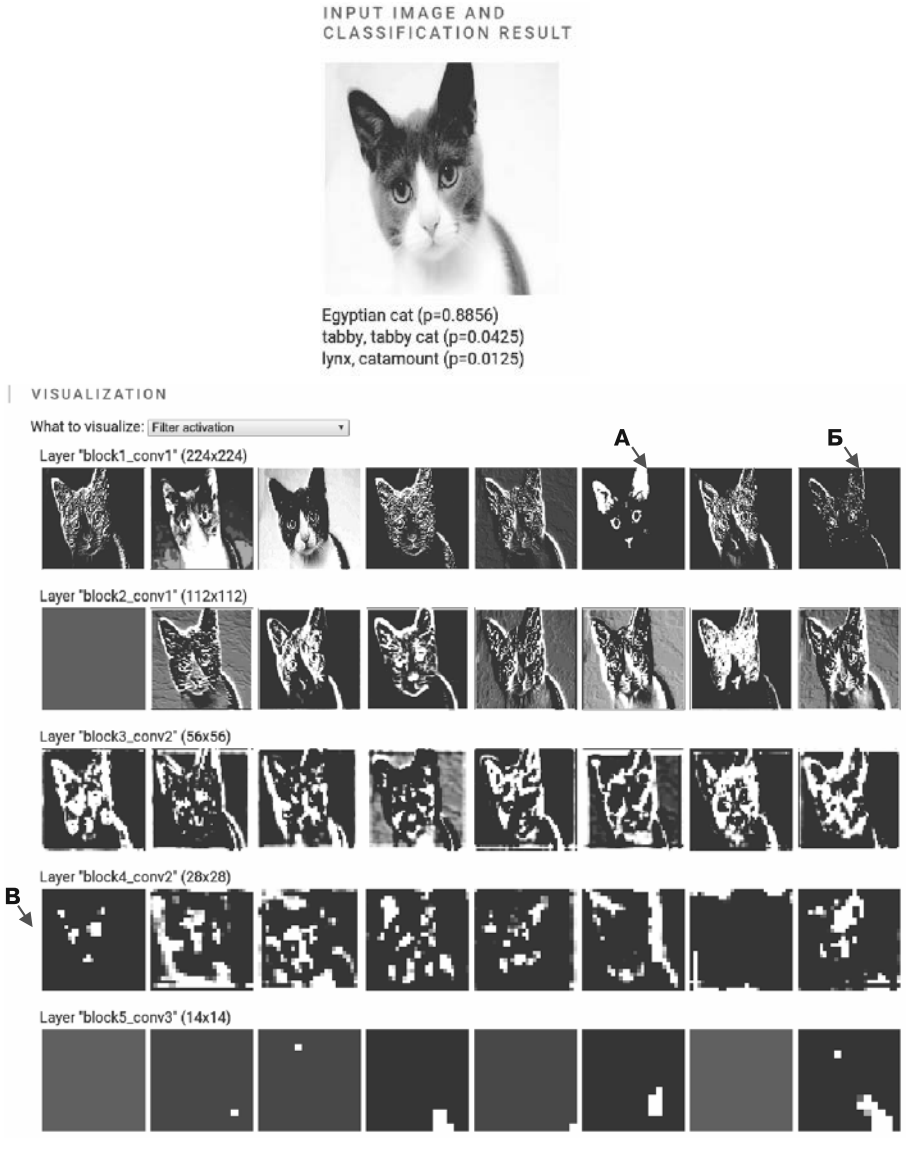


Рис. 7.8. Внутренние активации нескольких сверточных слоев модели VGG16 во время выполнения вывода на основе изображения cat.jpg. Исходное входное изображение приведено слева вместе с тремя наиболее вероятными классами, выведенными моделью, и соответствующими оценками вероятности. Здесь визуализированы пять слоев, а именно: block1_conv1, block2_conv1, block3_conv2, block4_conv2 и block5_conv3. Они упорядочены сверху вниз по глубине их расположения в модели VGG16, то есть слой block1_conv1 расположен ближе всего к входному слою, а block5_conv1 — к выходному. Учтите, что все изображения внутренних активаций для визуализации масштабируются к одному размеру, хотя размеры активаций (разрешение изображений) в более поздних слоях меньше из-за последовательной свертки и субдискретизации. Это заметно по более грубым паттернам пикселов в более поздних слоях

Подробности извлечения внутренних активаций

Код выделения внутренних активаций мы заключили в функцию `writeInternalActivationAndGetOutput()` (листинг 7.8). На входе она принимает уже сформированный нами или загруженный объект модели TensorFlow.js, а также названия интересующих нас слоев (`layerNames`). Ключевой шаг — создание нового объекта модели (`compositeModel`) с несколькими выходными сигналами, включая выходной сигнал указанных слоев и выходной сигнал исходной модели. `compositeModel` формируется с помощью API `tf.model()`, как вы видели в примерах Пакмана и simple-object-detection в главе 5. Объект `compositeModel` удобен тем, что его метод `predict()` возвращает все активации слоев вместе с итоговым предсказанием модели (см. константу `outputs`). Остальной код в листинге 7.8 (из файла `visualize-convnet/main.js`) выполняет более приземленные задачи разбиения выходных сигналов слоев на отдельные фильтры и запись их в файлы на диске.

Листинг 7.8. Вычисление внутренней активации сверточной сети в Node.js

```
async function writeInternalActivationAndGetOutput(
    model, layerNames, inputImage, numFilters, outputDir) {
  const layerName2FilePaths = {};
  const layerOutputs =
      layerNames.map(layerName => model.getLayer(layerName).output);
  const compositeModel = tf.model(
      {
        inputs: model.input,
       outputs: layerOutputs.concat(model.outputs[0])
      });
  const outputs = compositeModel.predict(inputImage);

  for (let i = 0; i < outputs.length - 1; ++i) {
    const layerName = layerNames[i];
    const activationTensors =
        tf.split(outputs[i],
                 outputs[i].shape[outputs[i].shape.length - 1],
                 -1);
    const actualNumFilters = filters <= activationTensors.length ?
        numFilters :
        activationTensors.length;
    const filePaths = [];
    for (let j = 0; j < actualNumFilters; ++j) {
      const imageTensor = tf.tidy(
          () => deprocessImage(tf.tile(activationTensors[j],
                                       [1, 1, 1, 3])));
      const outputFilePath = path.join(
          outputDir, `${layerName}_${j + 1}.png`);
      filePaths.push(outputFilePath);
      await utils.writeImageTensorToFile(imageTensor, outputFilePath);
    }
    layerName2FilePaths[layerName] = filePaths;
    tf.dispose(activationTensors);
  }
  tf.dispose(outputs.slice(0, outputs.length - 1));
  return {modelOutput: outputs[outputs.length - 1], layerName2FilePaths};
}
```

Формируем модель, возвращающую все нужные нам внутренние активации, помимо итогового выходного сигнала исходной модели

outputs — массив объектов tf.Tensor, содержащий внутренние активации и итоговый выходной сигнал

Разбиваем активацию сверточного слоя по фильтру

Приводим тензоры активаций к нужному формату и записываем их на диск

7.2.2. Визуализируем именно то, к чему чувствительны сверточные слои: наиболее активирующие изображения

Еще один способ показать, что усвоила сверточная сеть: найти входные изображения, к которым чувствительны ее разнообразные внутренние слои. Под фразой «фильтр чувствителен к определенному входному изображению» мы понимаем максимальную активацию выходного сигнала фильтра (усредненного по выходным измерениям высоты и ширины) при этом входном изображении. Изучение подобных наиболее активирующих входных сигналов для различных слоев сверточной сети позволяет выяснить, на что именно обучился реагировать каждый из слоев.

А найдем мы наиболее активирующие изображения с помощью уловки, ставящей «нормальный» процесс обучения нейронной сети с ног на голову. В блоке А на рис. 7.9 схематически показано, что происходит при обучении нейронной сети с помощью вызова `tf.Model.fit()`. Мы блокируем входные данные и позволяем весовым коэффициентам модели (ядрам и смещениям всех обучаемых слоев) обновляться на основе функции потерь[1] путем обратного распространения ошибки. Однако ничто не запрещает нам поменять местами роли входного сигнала и весовых коэффициентов: заблокировать весовые коэффициенты, позволив *входному сигналу* обновляться путем обратного распространения ошибки. Тем временем мы подправляем функцию потерь так, чтобы обратное распространение ошибки «подталкивало» входной сигнал в сторону максимизации выходного сигнала конкретного сверточного фильтра при усреднении по измерениям высоты и ширины.

Данный процесс изображен схематически в блоке Б на рис. 7.9 и называется *градиентным подъемом в пространстве входных данных* (gradient ascent in input space), в отличие от *градиентного спуска в пространстве весовых коэффициентов* (gradient descent in weight space), лежащего в основе обычного обучения модели. Код, реализующий градиентный подъем в пространстве входных данных, приведен в следующем разделе.

На рис. 7.10 показан результат выполнения процесса градиентного подъема в пространстве входных данных для четырех сверточных слоев модели VGG16 (той самой, на которой мы демонстрировали внутренние активации). Как и на предыдущей иллюстрации, глубина расположения слоев в модели растет сверху вниз. Из этих наиболее активирующих входных изображений можно почерпнуть несколько интересных паттернов.

- Во-первых, это цветные изображения, а не внутренние активации в оттенках серого, как в предыдущем разделе. А все потому, что их формат соответствует формату фактического входного сигнала сверточной сети: изображение, состоящее из трех (RGB) каналов. Потому их можно отобразить в цвете.

[1] Эту схему можно считать упрощенной версией рис. 2.9, на котором мы знакомили вас с обратным распространением ошибки в главе 2.

Рис. 7.9. Схематичная иллюстрация главной идеи поиска наиболее активирующего изображения для сверточного фильтра путем градиентного подъема в пространстве входных данных (блок Б) и отличия этой методики от обычного процесса обучения нейронной сети, в основе которого лежит градиентный спуск в пространстве весовых коэффициентов (блок А). Обратите внимание: этот рисунок отличается от некоторых предыдущих схем моделей — весовые коэффициенты вынесены из модели, чтобы подчеркнуть два отдельных набора величин, которые можно обновлять путем обратного распространения ошибки: весовые коэффициенты и входной сигнал

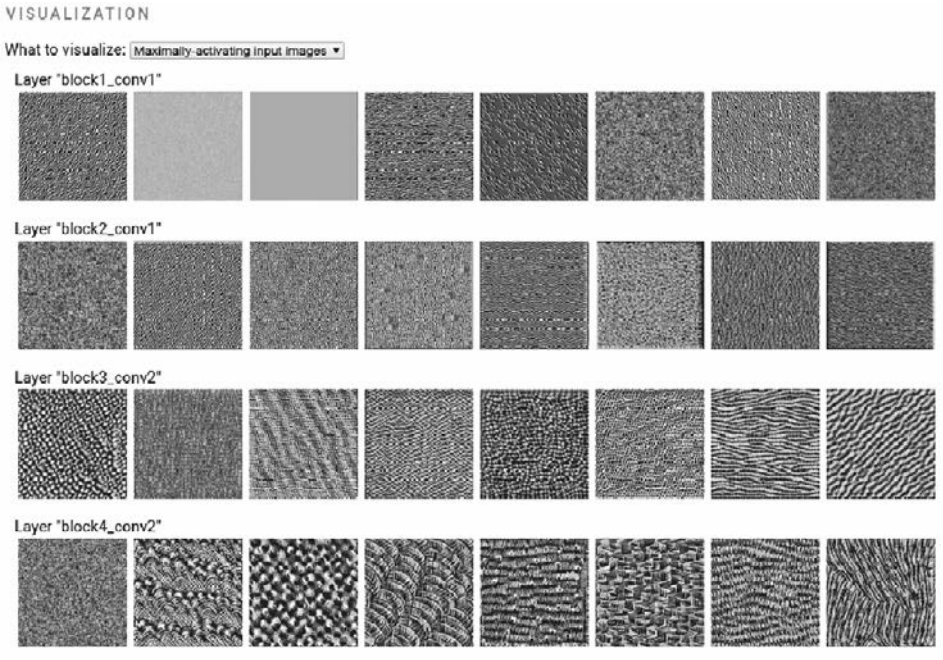


Рис. 7.10. Наиболее активирующие входные изображения для четырех слоев глубокой сверточной сети VGG16, вычисленные в ходе 80 итераций градиентного подъема в пространстве входных данных

- Самый первый слой (`block1_conv1`) чувствителен к простым паттернам, например к общим значениям цветов и границам фрагментов с определенной ориентацией.
- Слои, расположенные на средней глубине в сети (например, `block2_conv1`), сильнее всего реагируют на простые текстуры, основанные на сочетании различных паттернов границ фрагментов.
- Фильтры в более глубоко расположенных слоях начинают реагировать на более сложные паттерны, напоминающие визуальные признаки на реальных изображениях (из обучающих данных ImageNet, конечно), например зернистость, впадины, цветные полосы, завитки, волны и т. д.

Вообще говоря, по мере углубления в модель паттерны все дальше отходят от уровня отдельных пикселов и становятся все более сложными и масштабными, отражая то, как глубокая нейронная сеть уровень за уровнем «дистиллирует» признаки, составляя паттерны паттернов. И хотя уровни абстракции фильтров одного слоя одинаковы, нюансы паттернов могут существенно различаться, подчеркивая, что каждый из слоев формирует несколько взаимно дополняющих представлений одного и того же входного сигнала, дабы захватить как можно больше полезной информации в целях решения поставленной перед сетью задачи.

Заглянем глубже в метод градиентного подъема в пространстве входных данных

Основная логика градиентного подъема в пространстве входных данных в примере visualize-convnet заключена в функции `inputGradientAscent()` из файла `main.js` и показана в листинге 7.9. Код выполняется в Node.js, поскольку по природе своей требует значительных ресурсов в смысле памяти и затрачиваемого времени[1]. Отметим, что хотя основная идея градиентного подъема в пространстве входных данных аналогична обучению модели на основе градиентного спуска в пространстве весовых коэффициентов (см. рис. 7.10), напрямую переиспользовать `tf.Model.fit()` мы не можем, поскольку эта функция ориентирована на блокирование входного сигнала и обновление весовых коэффициентов. Вместо этого нам придется описать пользовательскую функцию вычисления «потерь» для заданного входного изображения. Эта функция описывается такой строкой кода:

```
const lossFunction = (input) =>
        auxModel.apply(input, {training: true}).gather([filterIndex], 3);
```

`auxModel` здесь — объект вспомогательной модели, созданный нами с помощью привычной функции `tf.model()`. У него тот же входной сигнал, что и у исходной модели, но на выходе — активация заданного сверточного слоя. Мы вызываем метод `apply()` вспомогательной модели, чтобы получить значение активации слоя. `apply()` аналогичен методу `predict()` в том, что осуществляет прямой проход модели. Впрочем, `apply()` обеспечивает более тонкие возможности управления, например позволяет задать значение `true` опции `training`, как в предыдущей строке кода. Без значения `true` опции `training` обратное распространение ошибки невозможно, поскольку при прямом проходе выделяемая под промежуточные активации память в конце освобождается для большей эффективности ее использования. Метод `apply()` при значении `true` флага `training` сохраняет эти внутренние активации, что дает возможность выполнять обратное распространение ошибки. А вызов `gather()` извлекает активацию конкретного фильтра. Это необходимо, поскольку наиболее активирующий входной сигнал вычисляется по каждому фильтру отдельно и результаты различаются для разных фильтров даже одного слоя (см. результаты примера на рис. 7.10).

Мы передаем нашу специальную функцию потерь в `tf.grad()` и получаем функцию, возвращающую градиент функции потерь по входному сигналу:

```
const gradFunction = tf.grad(lossFunction);
```

Важно понимать, что `tf.grad()` возвращает не сами значения градиента, а функцию (`gradFunction` в предыдущей строке кода), которая уже возвращает значения градиента.

---

[1] В случае меньших, чем VGG16, сверточных сетей (например, MobileNet и MobileNetV2) этот алгоритм может отработать за приемлемое время и в браузере.

Мы вызываем эту функцию градиента в цикле. На каждой итерации цикла обновляем входное изображение на основе возвращаемого ею значения градиента. Важный неочевидный нюанс: необходимо нормализовать значения градиента, прежде чем прибавлять их к входному изображению, гарантируя тем самым сопоставимые величины обновлений на всех итерациях:

```
const norm = tf.sqrt(tf.mean(tf.square(grads))).add(EPSILON);
return grads.div(norm);
```

В итоге выполнения этого итеративного обновления входного изображения 80 раз получаем приведенные на рис. 7.10 результаты.

Листинг 7.9. Градиентный подъем в пространстве входных данных (в Node.js, из файла visualize-convnet/main.js)

```
function inputGradientAscent(
    model, layerName, filterIndex, iterations = 80) {
  return tf.tidy(() => {
    const imageH = model.inputs[0].shape[1];
    const imageW = model.inputs[0].shape[2];
    const imageDepth = model.inputs[0].shape[3];

    const layerOutput = model.getLayer(layerName).output;
    const auxModel = tf.model({
      inputs: model.inputs,
     outputs: layerOutput
    });

    const lossFunction = (input) =>
        auxModel.apply(input, {training: true}).gather([filterIndex], 3);

    const gradFunction = tf.grad(lossFunction);

    let image = tf.randomUniform([1, imageH, imageW, imageDepth], 0, 1)
                  .mul(20).add(128);

    for (let i = 0; i < iterations; ++i) {
      const scaledGrads = tf.tidy(() => {
        const grads = gradFunction(image);
        const norm = tf.sqrt(tf.mean(tf.square(grads))).add(EPSILON);
        return grads.div(norm);
      });
      image = tf.clipByValue(
          image.add(scaledGrads), 0, 255);
    }
    return deprocessImage(image);
  });
}
```

Создает вспомогательную модель, входной сигнал у которой — такой же, как у исходной модели, а выходной сигнал — интересующий нас сверточный слой

Функция вычисляет значение выходного сигнала сверточного фильтра для заданного индекса фильтра

Генерирует случайное изображение в качестве отправной точки градиентного подъема

Важная уловка: масштабируем градиент по величине (норме) градиента

Выполняем очередной шаг градиентного подъема: обновляем изображение по направлению градиента

Функция вычисляет градиент выходного сигнала сверточного фильтра относительно входного изображения

7.2.3. Визуальная интерпретация результата классификации изображения сверточной сетью

Последняя из методик визуализации после обучения сверточной сети, с которой мы вас познакомим: алгоритм *карты активации классов* (class activation map, CAM). CAM стремится ответить на вопрос: «Какие элементы входного изображения играют главную роль в выводимом сверточной сетью наиболее вероятном варианте классификации?» Например, при передаче сети VGG16 изображения `cat.jpg` она выдает в качестве наиболее вероятного класса Egyptian cat («Египетский мау») — с вероятностью 0,89. Но по самому входному изображению и результатам классификации сложно сказать, какие части изображения оказались более важными для принятия сетью этого решения. Безусловно, определенные части изображения (например, голова кошки) сыграли более важную роль, чем прочие (например, белый фон). Но существует ли объективный способ выразить это количественно для любого входного изображения?

Ответ: да! Существует несколько способов, один из которых — использовать CAM[1]. На основе входного изображения и результата классификации сверточной сети CAM позволяет получить карту оценок важности различных частей этого изображения. На рис. 7.11 приведены подобные сгенерированные CAM карты интенсивности, наложенные поверх трех входных изображений: кошки, совы и двух слонов. Как видим из результата для кошки, наиболее высокие значения в карте интенсивности — у очертаний головы кошки. Постфактум можно сделать вывод: дело в том, что эти очертания демонстрируют форму головы животного — характерный отличительный признак для кошки. Карт интенсивности для совы также

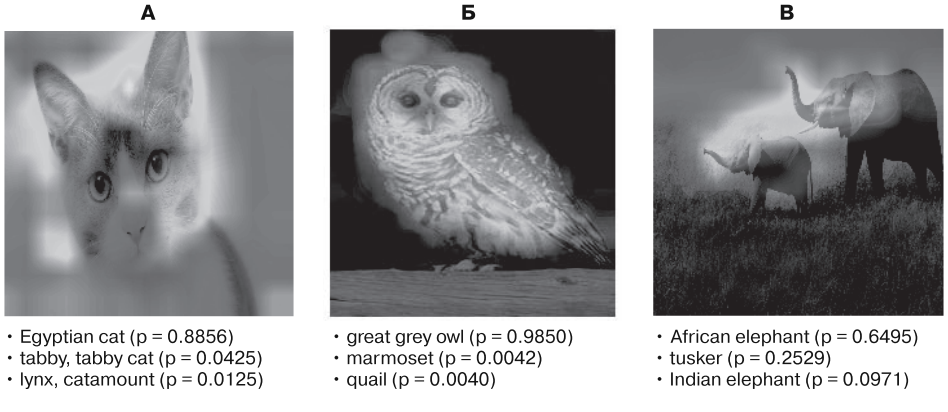


Рис. 7.11. Карты активаций классов (CAM) для трех входных изображений глубокой нейронной сети VGG16. Карты интенсивности CAM наложены на исходные входные изображения

[1] Алгоритм CAM впервые был описан в статье: *Zhou B. et al.* Learning Deep Features for Discriminative Localization. 2016. http://cnnlocalization.csail.mit.edu/. Еще один широко известный метод: локально интерпретируемые объяснения, не зависящие от устройства модели (Local Interpretable Model-Agnostic Explanations, LIME). См. http://mng.bz/yzpq.

соответствует ожидаемому: на ней выделены голова и крыло птицы. Интересен результат для изображения двух слонов, поскольку оно, в отличие от двух других изображений, включает два животных вместо одного. В сгенерированной алгоритмом CAM карте интенсивности наиболее высокие показатели важности — в области голов обоих слонов на изображении. Карта интенсивности явно фокусируется на бивнях и ушах животных, видимо, потому, что по ним можно отличить африканских слонов (наиболее вероятный класс с точки зрения сети) от индийских (третий по вероятности класс).

Техническая сторона алгоритма CAM

Несмотря на потрясающие возможности алгоритма CAM, лежащая в его основе идея довольно проста. По существу, каждый пиксел карты CAM показывает, насколько изменится вероятность класса-победителя, если значение пиксела увеличится на одну единицу измерения. Если подробнее, алгоритм CAM включает следующие шаги.

1. Найти последний (то есть расположенный глубже всего) сверточный слой сети. В VGG16 этот слой называется `block5_conv3`.
2. Вычислить градиент выходной вероятности сети для класса-победителя относительно выходного сигнала сверточного слоя.
3. Форма этого градиента: `[1, h, w, numFilters]`, где `h`, `w` и `numFilters` — высота, ширина и количество фильтров выходного сигнала слоя соответственно. Далее мы усредняем этот градиент по измерениям примеров данных, высоты и ширины, получая тензор формы `[numFilters]`, представляющий собой массив показателей важности, по одному для каждого фильтра сверточного слоя.
4. Умножаем тензор показателей важности (формы `[numFilters]`) на фактическое значение выходного сигнала сверточного слоя (формы `[1, h, w, numFilters]`) с помощью транслирования (см. подраздел B.2.2). В результате получаем новый тензор формы `[1, h, w, numFilters]` — масштабированную по важности версию выходного сигнала слоя.
5. Наконец, усредняем масштабированный по важности выходной сигнал слоя по последнему измерению (измерению фильтров) и «выжимаем» первое измерение (измерение примеров данных), в результате чего получаем изображение в оттенках серого формы `[h, w]`. Значения в этом изображении отражают, насколько важна соответствующая часть изображения для «победившего» результата классификации. Впрочем, это изображение включает отрицательные значения и размеры его измерений меньше, чем у исходного входного изображения (14×14 вместо 224×224 в нашем примере VGG16). Поэтому мы обнуляем отрицательные значения и интерполируем изображение перед наложением его на исходное входное изображение.

Подробный код можно найти в функции `gradClassActivationMap()` в файле `visualizeconvnet/main.js`. Хотя по умолчанию эта функция выполняется в Node.js, объем требуемых вычислений значительно меньше, чем в алгоритме градиентного подъема в пространстве входных данных из предыдущего раздела. Так что можно

запустить алгоритм CAM с помощью того же кода и в браузере со вполне приемлемым быстродействием.

В этой главе мы обсуждали два вопроса: визуализацию данных до попадания их в модель машинного обучения и визуализацию модели после ее обучения. Мы умышленно опустили важный промежуточный этап — визуализацию модели *во время* ее обучения. Этому посвящена следующая глава. Мы выделили этот вопрос в отдельную главу потому, что он тесно связан с понятиями и явлениями недообучения и переобучения, критически важными для всех задач обучения с учителем, а потому заслуживающими отдельного обсуждения. Визуализация значительно упрощает обнаружение и исправление недообучения и переобучения. В следующей главе мы вновь обратимся к библиотеке tfjs-vis, с которой познакомили вас в начале текущей главы, и покажем, чем она может помочь в демонстрации хода процесса обучения модели, помимо обсуждавшихся ее возможностей визуализации данных.

Материалы для дальнейшего изучения

- *Ribeiro M. T., Singh S., Guestrin C.* Why Should I Trust You? Explaining the Predictions of Any Classifier. 2016. https://arxiv.org/pdf/1602.04938.pdf.
- Во фреймворке TensorSpace (tensorspace.org) — надстройке над TensorFlow.js, `three.js` и `tween.js` — для визуализации топологии и внутренних активаций сверточных сетей в браузере используется трехмерная компьютерная графика.
- Библиотека t-SNE TensorFlow.js (https://github.com/tensorflow/tfjs-tsne) представляет собой основанную на WebGL эффективную реализацию алгоритма стохастических вложений соседей на основе распределения Стьюдента. С ее помощью можно визуализировать многомерные наборы данных путем проекции их в двумерное пространство с сохранением важных структур данных.

Упражнения

1. Поэкспериментируйте со следующими возможностями функции `tfjs.vis.linechart()`.
   А. Модифицируйте код из листинга 7.2 и посмотрите, что получится при построении графиков двух рядов данных с различными наборами значений координаты *x*. Например, попробуйте взять для первого ряда данных значения 1, 3, 5 и 7 координаты *x*, а для второго — 2, 4, 6 и 8. Можете создать ветку блокнота CodePen, расположенного по адресу https://codepen.io/tfjs-book/pen/BvzMZr.
   Б. Все линейные диаграммы в примере CodePen созданы на основе рядов данных, у которых значения по оси *X* не дублируются. Выясните, как функция `linechart()` работает с точками данных, у которых значения по оси *X* совпадают. Например, включите в ряд данных две точки данных со значением 0 по оси *X*, но различными значениями по оси *Y* (например, –5 и 5).

2. В примере visualize-convnet задайте собственное входное изображение с помощью флага `--image` команды `yarn visualize`. А поскольку в разделе 7.2 мы использовали только изображения животных, попробуйте другие типы содержимого изображений, например людей, машин, предметов быта и природы. Посмотрите, какую полезную информацию можно почерпнуть из соответствующих внутренних активаций и карт CAM.
3. В примере, где мы рассчитывали CAM модели VGG16, вычислялись градиенты показателей вероятности для класса-*победителя* по выходному сигналу последнего сверточного слоя. Что, если вместо этого вычислить градиенты класса, который *не является «победителем»* (класса с меньшей вероятностью)? Следует ожидать, что в получившемся изображении CAM *не* будут выделяться ключевые фрагменты, относящиеся к настоящему объекту изображения. Убедитесь в этом, модифицировав код примера visualize-convnet и запустив его снова. А именно, передавайте индекс класса для вычисления градиентов в виде аргумента функции `gradClassActivationMap()` из файла `visualize-convnet/cam.js`. Эта функция вызывается в файле `visualize-convnet/main.js`.

Резюме

- Мы изучили основы использования tfjs-vis, библиотеки визуализации, тесно интегрированной с TensorFlow.js и пригодной для визуализации основных типов графиков в браузере.
- Визуализация данных — неотъемлемая составляющая машинного обучения. Эффективное визуальное представление данных позволяет выявить паттерны и почерпнуть полезную информацию о них, которую иначе добыть было бы очень непросто, как мы продемонстрировали на данных Jena-weather-archive.
- Из уже обученных нейронных сетей можно найти немало интересных паттернов и почерпнуть много полезной информации. Мы показали основные этапы и результаты.
  - Визуализации внутренних активаций глубокой сверточной сети.
  - Вычисления элементов входного сигнала, на которые слои реагируют сильнее всего.
  - Выяснения того, какие части входного изображения играют главную роль в принятии сверточной сетью решения о классификации. Это помогает понять, что усвоила сверточная сеть и как она функционирует во время выполнения вывода.

8 Недообучение, переобучение и универсальный технологический процесс машинного обучения

В этой главе

- Почему так важно визуализировать процесс обучения модели и на что следует обратить внимание.
- Как визуализировать недообучение и переобучение и понять, что к чему.
- Основной способ решения проблемы переобучения: регуляризация и визуализация ее эффекта.
- Что представляет собой универсальный технологический процесс машинного обучения, из каких шагов он состоит и почему этим важным набором инструкций руководствуются все задачи машинного обучения с учителем.

В предыдущей главе вы научились использовать tfjs-vis для визуализации данных перед проектированием и обучением на них моделей МО. В этой главе мы продолжим с места, на котором завершили предыдущую, и опишем, как использовать tfjs-vis для визуализации структуры и метрик моделей во время их обучения. Основная наша цель — вовремя обнаружить чрезвычайно важные явления *недообучения* (underfitting) и *переобучения* (overfitting). Научившись их обнаруживать, мы займемся их устранением и с помощью визуализации проверкой того, что наши методики их устранения работают.

8.1. Постановка задачи предсказания температуры

Для демонстрации недообучения и переобучения нам нужна конкретная задача машинного обучения. Воспользуемся задачей предсказания температуры на основе данных из набора Jena-weather, который вы встретили в предыдущей главе. В разделе 7.1 на наборе данных Jena-weather демонстрировались возможности и преимущества визуализации данных в браузере. Надеемся, вы прочувствовали этот набор данных, поэкспериментировав с UI визуализации. Теперь мы готовы приступить к применению машинного обучения к этому набору данных. Но сначала необходимо сформулировать задачу.

Этот пример можно рассматривать как «игрушечную» задачу прогноза погоды. Мы попытаемся в ней предсказать температуру за 24 часа, следующие за определенным моментом времени, на основе 14 типов метеорологических измерений, полученных за предшествующий этому моменту десятидневный период.

И хотя формулировка задачи проста, способ генерации данных из CSV-файла требует определенных пояснений, поскольку отличается от процедур генерации данных в предыдущих задачах книги. В них каждая строка исходного файла данных соответствовала обучающему примеру. Именно так были устроены примеры iris-flower, Boston-housing и phishing-detection (см. главы 2 и 3). В этой же задаче каждый пример данных формируется путем выборки и сочетания нескольких строк CSV-файла, поскольку температура предсказывается на основе данных за определенный промежуток времени, а не за один конкретный момент (рис. 8.1).

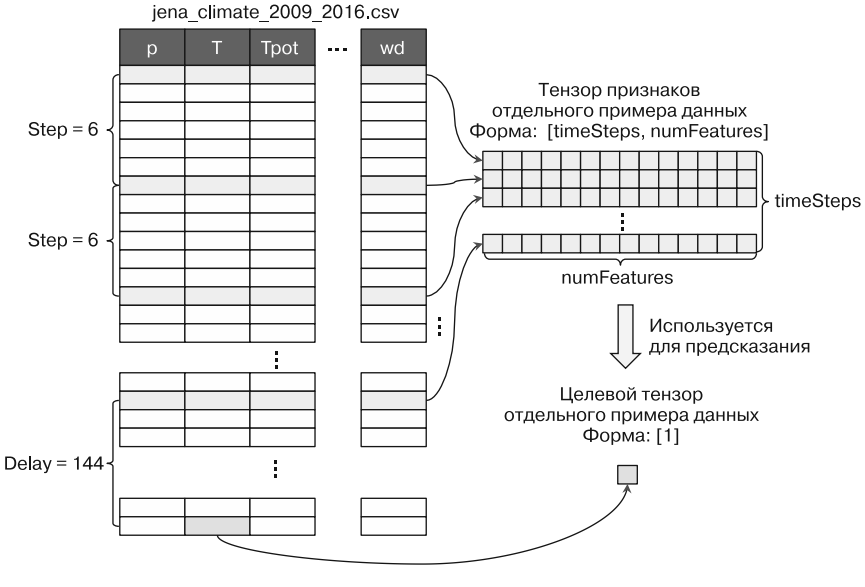


Рис. 8.1. Схематическое изображение генерации отдельного примера на основе табличных данных

Разберем рис. 8.1. Для генерации тензора признаков примера данных производится выборка из CSV-файла через каждые `step` строк (для примера возьмем `step` = 6), всего `timeSteps` строк (для примера `timeSteps` = 240). В результате получается тензор формы `[timeSteps, numFeatures]`, где `numFeatures` (по умолчанию — 14) представляет собой количество столбцов признаков в CSV-файле. Для генерации целевого тензора производим выборку значения температуры (`T`), начиная с шага, соответствующего строке `delay` (для примера — 144), после последней строки, попавшей в тензор признаков. Прочие примеры данных можно сгенерировать, начав с другой строки CSV-файла, но они формируются по тому же правилу. В результате получаем следующую задачу предсказания температуры: по заданным 14 метеорологическим измерениям за указанный период времени (например, десять дней), предшествующий текущему моменту, предсказать температуру на момент, отстоящий от текущего на `delay` (например, 24 часа). Показанные на этой схеме действия производит код функции `getNextBatchFunction()` из файла `jena-weather/data.js`.

Для генерации признаков обучающего примера мы выбираем набор строк за промежуток времени десять дней. Вместо того чтобы использовать все строки данных за эти десять дней, мы выбираем только каждую шестую строку. Почему? По двум причинам. Во-первых, при выборке всех строк получилось бы в шесть раз больше данных, что означает модель большего размера и более длительное обучение. Во-вторых, избыточность данных при часовой шкале времени довольно велика (атмосферное давление шесть часов назад обычно мало отличалось от давления шесть часов и десять минут назад). Благодаря отбрасыванию пяти шестых частей данных мы получаем более эффективную модель меньшего размера, практически не теряя ее способности к предсказанию. Выбранные строки объединяются в двумерный тензор признаков формы `[timeSteps, numFeatures]` для нашего обучающего примера. По умолчанию значение `timeSteps` равно 240, что соответствует 240 моментам выборки, равномерно распределенным по промежутку времени десять дней. Значение `numFeatures` равно 14, что соответствует показаниям 14 метеорологических приборов в CSV наборе данных.

Получить целевой тензор для обучающего примера проще: необходимо лишь перейти вперед на определенный промежуток времени (`delay`) от последней строки, включаемой в тензор признаков, и взять значение из столбца температуры. На рис. 8.1 показана генерация только одного обучающего примера. Для генерации нескольких примеров данных необходимо просто начинать с различных строк CSV-файла.

Вы могли заметить нечто необычное в тензоре признаков для нашей задачи предсказания температуры (см. рис. 8.1): во всех предыдущих задачах тензор признаков отдельного примера данных был одномерным, в результате чего при организации нескольких примеров данных в батч получался двумерный тензор. Однако в данной задаче тензор признаков отдельного примера уже двумерный, а значит, при объединении нескольких примеров в батч получится трехмерный тензор (формы `[batchSize, timeSteps, numFeatures]`). Тонкое наблюдение! Возникает такая двумерная форма тензора признаков из-за того, что источник признаков — *последовательность* событий. В частности, метеорологические измерения за 240 моментов времени. Этим данная задача отличается от всех прочих встречавшихся вам до сих пор в книге задач, в которых входные признаки для конкретного примера не

охватывали несколько моментов времени — ни измерения размеров цветов в задаче iris-flower, ни значения 28×28 пикселов изображения MNIST[1].

Можно сказать, сейчас вы в первый раз в книге встретились с последовательными входными данными. В следующей главе мы более подробно обсудим вопрос создания в TensorFlow.js специализированных моделей (RNN) для последовательных данных, обладающих более широкими возможностями. Но здесь воспользуемся для поставленной задачи двумя уже известными вам типами моделей: линейными регрессорами и многослойными перцептронами. Таким образом мы сформируем фундамент для изучения RNN и получим точку отсчета для сравнения с более продвинутыми моделями.

Сам код, выполняющий показанный на рис. 8.1 процесс генерации данных, находится в файле `jena-weather/data.js`, в функции `getNextBatchFunction()`. Она интересна тем, что возвращает объект с функцией `next()` вместо конкретного значения. Функция `next()` при вызове возвращает фактические значения данных. А содержащий функцию `next()` объект называется *итератором*. Зачем использовать подобный обходной путь вместо написания непосредственно итератора? Во-первых, это соответствует спецификации «генератор/итератор» языка JavaScript[2]. Вскоре мы передадим его в API `tf.data.generator()`, чтобы создать объект `Dataset` для обучения модели. А этот API требует именно такой сигнатуры функции. Во-вторых, нам нужна возможность задавать настройки итератора; а возвращающая итератор функция — отличное средство, позволяющее задавать конфигурации.

Возможные опции конфигурации видны из сигнатуры функции `getNextBatchFunction()`:

```
getNextBatchFunction(
      shuffle, lookBack, delay, batchSize, step, minIndex, maxIndex,
          normalize,
      includeDateTime)
```

Как видим, параметров настройки здесь немало. Например, с помощью аргумента `lookBack` можно задавать длительность анализируемого периода при прогнозе погоды. А аргумент `delay` позволяет указать, насколько далеко в будущее простирается этот прогноз. Аргументы `minIndex` и `maxIndex` позволяют задавать диапазон строк, из которых выбираются данные, и т. д.

Передавая функцию `getNextBatchFunction()` функции `tf.data.generator()`, мы преобразуем ее в объект `tf.data.Dataset`. Как мы упоминали в главе 6, совместное использование объекта `tf.data.Dataset` с методом `fitDataset()` объекта `tf.Model` дает возможность обучать модель даже в случае, когда данные не помещаются целиком в память WebGL (или любой другой подходящий тип памяти). Объект `Dataset` создает батч обучающих данных в GPU лишь непосредственно перед обучением.

---

[1] На самом деле в задаче распознавания голосовых команд в главе 4 присутствовала последовательность событий, а именно последовательные кадры аудиоспектра, составляющие спектрограмму. Впрочем, мы рассматривали спектрограмму в целом как изображение, игнорируя временно́е измерение данной задачи и работая с ним как с пространственным.

[2] См. веб-документацию MDN, раздел Iterators and Generators («Итераторы и генераторы»): http://mng.bz/RPWK.

Именно так мы и поступим с задачей предсказания температуры. На самом деле мы не смогли бы обучить модель с помощью обычного метода `fit()` модели из-за большого количества и размера примеров данных. Код вызова `fitDataset()` можно найти в файле `jena-weather/models.js`, и выглядит он так, как показано в листинге 8.1.

Листинг 8.1. Визуализация обучения модели с помощью tfjs-vis на основе метода fitDataset

```
const trainShuffle = true;
const trainDataset = tf.data.generator(
    () => jenaWeatherData.getNextBatchFunction(
      trainShuffle, lookBack, delay, batchSize, step, TRAIN_MIN_ROW,
      TRAIN_MAX_ROW, normalize, includeDateTime)).prefetch(8);
const evalShuffle = false;
const valDataset = tf.data.generator(
  () => jenaWeatherData.getNextBatchFunction(
      evalShuffle, lookBack, delay, batchSize, step, VAL_MIN_ROW,
      VAL_MAX_ROW, normalize, includeDateTime));

  await model.fitDataset(trainDataset, {
  batchesPerEpoch: 500,
  epochs,
  callbacks: customCallback,
  validationData: valDataset
});
```

Первый объект Dataset сгенерирует обучающие данные

Второй объект Dataset сгенерирует проверочные данные

Поле validationData может принимать либо объект Dataset, либо набор тензоров. Здесь используется первый вариант

Первые два поля объекта конфигурации метода `fitDataset()` задают количество эпох обучения модели и количество батчей для каждой эпохи. Как вы уже знаете из главы 6, это обычные поля настроек вызова метода `fitDataset()`. Однако третье поле (`callbacks: customCallback`) нам пока не встречалось. Именно с его помощью мы собираемся визуализировать процесс обучения. `customCallback` может принимать различные значения в зависимости от того, происходит ли обучение модели в браузере или, как в следующей главе, в Node.js.

В браузере значение `customCallback` предоставляет функция `tfvis.show.fitCallbacks()`. Она позволяет визуализировать обучение модели на веб-странице с помощью всего одной строки кода на JavaScript. Она не только избавляет нас от работы по обращению к функции потерь и отслеживанию значений метрик для каждого батча и каждой эпохи, но и устраняет необходимость вручную создавать и поддерживать HTML-элементы для отрисовки графиков:

```
const trainingSurface =
    tfvis.visor().surface({tab: modelType, name: 'Model Training'});
 const customCallback = tfvis.show.fitCallbacks(trainingSurface,
    ['loss', 'val_loss'], {
   callbacks: ['onBatchEnd', 'onEpochEnd']
 }));
```

Первый аргумент функции `fitCallbacks()` задает область визуализации, создаваемую с помощью метода `tfvis.visor().surface()`. В принятой в tfjs-vis терминологии она называется *поверхностью визира* (visor surface). Визир — это контейнер для удобной организации всех элементов визуализации при машинном обучении в браузере. В структуре визира присутствует два уровня иерархии. Верхний вклю-

чает одну или несколько вкладок, по которым пользователь может перемещаться щелчками кнопкой мыши. На нижнем уровне каждая из вкладок содержит одну или несколько *поверхностей* (surfaces). Метод `tfvis.visor().surface()` с его полями конфигурации `tab` и `name` позволяет создавать поверхность с заданным названием на указанной вкладке визира. Поверхность визира не ограничивается отрисовкой кривых потерь и метрик. На самом деле отрисовать на поверхностях визиров можно все основные типы диаграмм, показанных в примере CodePen в разделе 7.1. Мы оставим проверку этого как упражнение вам в конце главы.

Второй аргумент функции `fitCallbacks()` задает, какие именно функции потерь и метрики будут отрисовываться на поверхности визира. В данном случае мы построим график функции потерь для обучающего и проверочного наборов данных. Третий аргумент содержит поле, служащее для управления частотой обновления графиков. При использовании обеих опций `onBatchEnd` и `onEpochEnd` графики будут обновляться в конце каждого из батчей и каждой из эпох. В следующем разделе мы изучим построенные функцией `fitCallbacks()` кривые потерь и продемонстрируем их использование для обнаружения недообучения и переобучения.

8.2. Недообучение, переобучение и меры противодействия им

Во время обучения модели МО желательно следить за тем, насколько хорошо она захватывает содержащиеся в обучающих данных паттерны. Модель, плохо захватывающая паттерны, считается *недообученной*; а модель, *слишком* хорошо захватывающая паттерны, до такой степени, что плохо обобщается на новые данные, — *переобученной*. Переобученную модель можно «вернуть на путь истинный» с помощью таких средств, как регуляризация. В этом разделе мы продемонстрируем выявление подобного поведения модели с помощью визуализации, а также эффект от мер противодействия ему.

8.2.1. Недообучение

Для решения задачи предсказания температуры попробуем сначала простейшую возможную модель машинного обучения: линейный регрессор. Такую модель создает код в листинге 8.2 (из файла `jenaweather/index.js`). Для генерации предсказания он использует плотный слой с одним нейроном и применяемую по умолчанию линейную функцию активации. Впрочем, в отличие от линейного регрессора, созданного нами для задачи предсказания времени скачивания в главе 2, у этой модели есть еще дополнительный слой схлопывания. Поскольку форма тензора признаков — двумерная, его необходимо схлопнуть в одномерный, чтобы получить подходящий входной сигнал для плотного слоя, используемого для линейной регрессии. Этот процесс приведен на рис. 8.2. Важно отметить, что при такой операции схлопывания теряется информация о последовательном (временно́м) упорядочении данных.

Листинг 8.2. Создание модели линейной регрессии для задачи прогноза температуры

```
function buildLinearRegressionModel(inputShape) {
  const model = tf.sequential();
  model.add(tf.layers.flatten({inputShape}));
  model.add(tf.layers.dense({units: 1}));
  return model;
}
```

Схлопываем форму [batchSize, timeSteps, numFeatures] входного тензора до [batchSize, timeSteps * numFeatures], чтобы удовлетворить требования плотного слоя

Плотный слой с одним нейроном, с функцией активации по умолчанию (линейной) представляет собой линейный регрессор

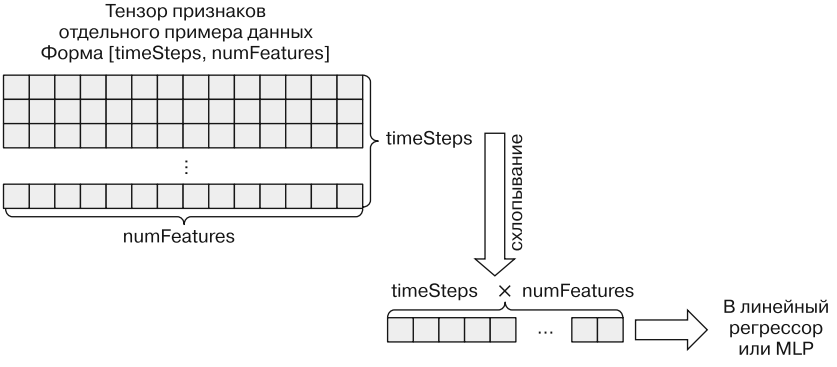


Рис. 8.2. Схлопывание двумерного тензора признаков формы [timeSteps, numFeatures] в одномерный тензор формы [timeSteps × numFeatures], производимое как линейным регрессором в листинге 8.2, так и моделью MLP в листинге 8.3

После создания модели мы компилируем ее для обучения с помощью команды:

```
model.compile({loss: 'meanAbsoluteError', optimizer: 'rmsprop'});
```

Здесь используется функция потерь `meanAbsoluteError`, поскольку мы прогнозируем непрерывную величину (нормализованную температуру). В отличие от некоторых из предыдущих задач никакой отдельной метрики не задано, поскольку функция потерь MAE сама является понятной человеку метрикой. Впрочем, учтите, что мы предсказываем *нормализованную* температуру, так что MAE для преобразования в абсолютную ошибку предсказания необходимо умножить на среднеквадратичное отклонение столбца температуры (8,476 градуса Цельсия). Например, при MAE 0,5 ошибка предсказания будет равна $8{,}476 \times 0{,}5 = 4{,}238$ градуса Цельсия.

В UI демонстрации в списке Model Type (Тип модели) выберите Linear Regression (Линейная регрессия) и нажмите Train Model (Обучить модель) для запуска процесса обучения линейного регрессора. Сразу после начала обучения вы увидите таблицу со сводкой топологии модели во всплывающей в правой части страницы «карточке» (рис. 8.3). Таблица сводных показателей модели чем-то напоминает выводимый `model.summary()` текст, но визуализируемый графически в HTML. Код создания этой таблицы выглядит следующим образом:

```
const surface = tfvis.visor().surface({name: 'Model Summary', tab});
tfvis.show.modelSummary(surface, model);
```

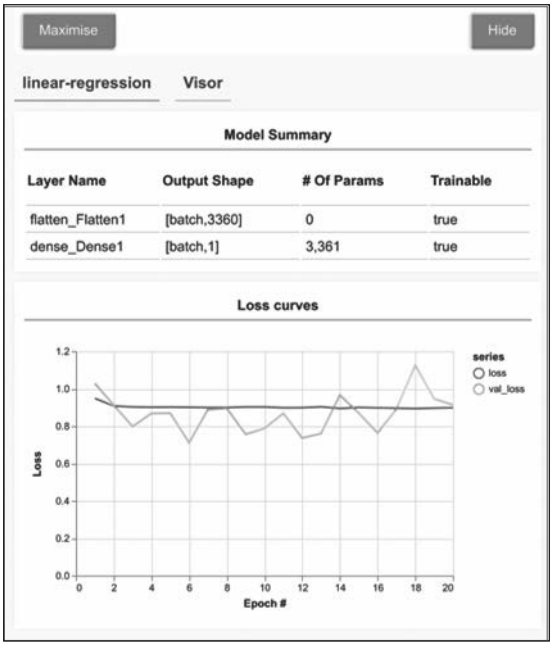


Рис. 8.3. Визир tfjs-vis, визуализирующий обучение модели линейной регрессии. *Вверху:* таблица со сводкой топологии модели. *Внизу:* кривые потерь после 20 эпох обучения. Диаграмма создана с помощью функции tfvis.show.fitCallbacks() (см. файл jena-weather/index.js)

После создания поверхности мы отрисовываем на ней таблицу со сводкой топологии модели, передавая эту поверхность методу `tfvis.show.modelSummary()`, как показано во второй строке предыдущего фрагмента кода.

В части Model Summary (Сведения о модели) вкладки linear-regression находится график, который отображает кривые потерь при обучении модели (см. рис. 8.3), создаваемый при вызове `fitCallbacks()`, описанном в предыдущем разделе. Из этого графика видно, насколько хорошо линейный регрессор решает задачу предсказания температуры. Потери как на обучающем, так и на проверочном наборах данных в конце колеблются в районе 0,9, что соответствует $8{,}476 \times 0{,}9 = 7{,}6$ градуса Цельсия в абсолютном выражении (напомним, что среднеквадратичное отклонение столбца температуры в CSV-файле составляет 8,476). Это значит, что после обучения ошибка предсказания нашего линейного регрессора составляет в среднем 7,6 градуса Цельсия (13,7 градуса Фаренгейта). Не слишком хорошее предсказание. Никто не стал бы доверять прогнозу погоды, основанному на этой модели! Это пример недообучения.

Недообучение обычно возникает при разрешающих возможностях модели, недостаточных для моделирования связи признака и целевой величины. В данном примере линейный регрессор слишком прост структурно, а потому не обладает достаточными возможностями для захвата зависимости метеорологических данных за предыдущие десять дней и температуры на следующий день. Для преодоления недообучения обычно повышают разрешающие возможности модели, увеличивая ее размер.

Типичные подходы к этому включают добавление в модель дополнительных слоев (с нелинейными функциями активации) и увеличение размера слоев (например, числа нейронов в плотном слое). Так что давайте добавим скрытый слой в наш линейный регрессор и посмотрим, насколько лучше станет работать получившийся MLP.

8.2.2. Переобучение

Создающая MLP-модели функция приведена в листинге 8.3 (из файла `jena-weather/index.js`). Создаваемый ею MLP включает два плотных слоя: один скрытый и один — выходной, помимо слоя схлопывания, предназначенного для той же цели, что и в модели линейной регрессии. Как видите, у этой функции на два аргумента больше, чем у функции `buildLinearRegressionModel()` в листинге 8.2. В частности, параметры `kernelRegularizer` и `dropoutRate` позднее помогут нам в борьбе с переобучением. А пока выясним, какой степени безошибочности предсказаний позволяет добиться MLP без `kernelRegularizer` и `dropoutRate`.

Листинг 8.3. Создание MLP для задачи предсказания температуры

```
function buildMLPModel(inputShape, kernelRegularizer, dropoutRate) {
  const model = tf.sequential();
  model.add(tf.layers.flatten({inputShape}));
  model.add(tf.layers.dense({
    units: 32,
    kernelRegularizer
    activation: 'relu',
  }));
  if (dropoutRate > 0) {
    model.add(tf.layers.dropout({rate: dropoutRate}));
  }
  model.add(tf.layers.dense({units: 1}));
  return model;
}
```

При задании этого параметра вызывающей стороной в ядро скрытого плотного слоя добавляется регуляризация

При задании этого параметра вызывающей стороной добавляется слой дропаута между скрытым и выходным плотными слоями

В блоке А на рис. 8.4 показаны кривые потерь при использовании MLP. Можно заметить несколько отличий от кривых потерь линейного регрессора.

- Кривые потерь для обучающего и проверочного наборов данных расходятся, в отличие от рис. 8.3, на котором они демонстрируют более или менее схожие тенденции.
- Кривая потерь на обучающем наборе данных сходится к намного меньшему значению погрешности, чем раньше. После 20 эпох обучения потери на обучающем наборе данных были около 0,2, что соответствует погрешности в $8{,}476 \times 0{,}2 = 1{,}7$ градуса Цельсия — намного лучше, чем результаты линейной регрессии.
- Впрочем, величина потерь на проверочном наборе данных лишь ненадолго уменьшается на первых двух эпохах, а затем снова понемногу растет. В конце 20-й эпохи она значительно выше, чем потери на обучающем наборе данных (0,35, то есть около 3 градусов Цельсия).

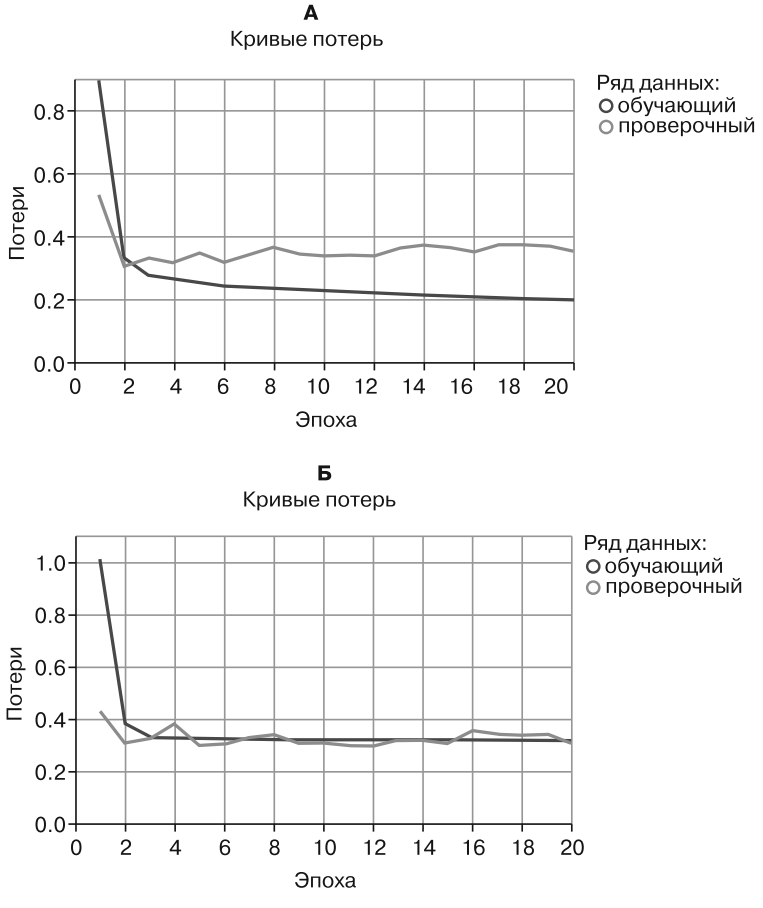


Рис. 8.4. Кривые потерь двух различных моделей на основе MLP для задачи предсказания температуры. Блок А: от модели MLP без какой-либо регуляризации. Блок Б: от модели MLP с теми же размерами и количеством слоев, что и модель в блоке А, но с L2-регуляризацией ядер плотных слоев. Обратите внимание, что диапазоны значений по оси Y слегка отличаются в двух графиках

Потери на обучающем наборе данных ниже более чем в четыре раза относительно предыдущего результата из-за больших возможностей нашего MLP по сравнению с моделью линейной регрессии, что достигается благодаря дополнительному слою и в несколько раз большему числу весовых параметров. Впрочем, есть у возросших возможностей модели и побочный эффект: модель оказывается подогнана к обучающим данным намного лучше, чем к проверочным (данным, которые модель не видела во время обучения). Это пример *переобучения* (overfitting), когда модель «уделяет слишком много внимания» несущественным деталям обучающих данных — до такой степени, что ее предсказания плохо обобщаются на не виденные ею данные.

8.2.3. Сокращаем переобучение за счет регуляризации весов и визуализируем эффект от нее

В главе 4 мы сокращали переобучение сверточной сети за счет добавления в модель слоев дропаута. Здесь мы рассмотрим еще один часто используемый подход к снижению переобучения: регуляризацию весовых коэффициентов. Если выбрать в UI демонстрации Jena-weather тип модели MLP with L2 Regularization (Многослойный перцептрон с L2-регуляризацией), то в коде будет создан MLP с помощью вызова `buildMLPModel()` (листинг 8.3):

```
model = buildMLPModel(inputShape, tf.regularizers.l2());
```

Второй аргумент — возвращаемое функцией `tf.regularizers.l2()` значение — представляет собой *L2-регуляризатор*. Соотнеся вышеприведенный код с функцией `buildMLPModel()` в листинге 8.3, вы увидите, что L2-регуляризатор попадает в параметр `kernelRegularizer` конфигурации скрытого плотного слоя, в результате чего применяется к ядру этого плотного слоя. А когда к весу (например, ядра плотного слоя) применяется регуляризатор, говорят, что он *регуляризован*. Аналогично, когда часть или все веса модели регуляризованы, говорят, что регуляризована модель в целом.

Так что же делает регуляризатор с ядром плотного слоя и MLP в целом? Он добавляет к функции потерь дополнительный член. Функция потерь нерегуляризованного MLP вычисляется просто как MAE между целями и предсказаниями модели. В псевдокоде это можно выразить следующим образом:

```
loss = meanAbsoluteError(targets, predictions)
```

При регуляризации весов функция потерь модели включает дополнительный член. В псевдокоде:

```
loss = meanAbsoluteError(targets, prediciton) + 12Rate * 12(kernel)
```

`12Rate * 12(kernel)` представляет собой дополнительный член L2-регуляризации функции потерь. В отличие от MAE этот член *не* зависит от предсказаний модели, а только от регуляризуемого ядра (весового коэффициента) слоя. Для заданного ядра он выдает число, связанное лишь со значениями этого ядра. Это число можно рассматривать как меру нежелательности текущего значения ядра.

Теперь рассмотрим подробное описание функции L2-регуляризации: `l2(kernel)`. Она вычисляет сумму квадратов всех значений весовых коэффициентов. Например, если ради простоты форма нашего ядра невелика: `[2, 2]`, а его значения — `[[0.1, 0.2], [-0.3, -0.4]]`, то:

```
l2(kernel) = 0.1^2 + 0.2^2 + (-0.3)^2 + (-0.4)^2 = 0.3
```

Следовательно, функция `l2(kernel)` всегда возвращает положительное число, делающее невыгодными большие значения весов в `kernel`. Этот член при включении его в общие потери поощряет меньшие по модулю значения элементов `kernel` при прочих равных условиях.

Теперь общие потери включают два различных члена: расхождение цели с предсказанием и член, отражающий величину значений `kernel`. В результате в процессе

обучения будет не только минимизироваться расхождение цели с предсказанием, но и уменьшаться сумма квадратов элементов ядра. Зачастую эти две задачи противоречат друг другу. Например, уменьшение величины элементов ядра может снижать второй член, но увеличивать первый (потери на основе MSE). Как же эти два противоречащих члена уравновешиваются по относительной значимости в общих потерях? Для этого и служит коэффициент `l2Rate`, выражающий количественно важность члена L2 относительно члена расхождения цели с предсказанием. Чем больше значение `l2Rate`, тем больше процесс обучения будет стремиться уменьшить член L2-регуляризации за счет увеличения погрешности цель — предсказание. Этот член, по умолчанию равный `1e-3`, представляет собой гиперпараметр, значение которого подбирается в ходе оптимизации гиперпараметров.

Так в чем же польза от L2-регуляризатора? В блоке Б на рис. 8.4 приведены кривые потерь регуляризованного MLP. Сравнивая их с кривыми для нерегуляризованного MLP (блок А того же рисунка), видим, что расхождение кривых потерь для обучающего и проверочного наборов данных у регуляризованной модели меньше. А значит, модель более не «уделяет слишком много внимания» специфическим паттернам обучающего набора данных. Вместо этого она усваивает из обучающего набора данных паттерны, хорошо обобщающиеся на не виденные моделью примеры данных из проверочного набора. В нашем регуляризованном MLP только первый плотный слой содержит регуляризатор, а второй плотный слой — нет. Но этого, как оказывается, в данном случае вполне достаточно для устранения переобучения. В следующем разделе мы подробнее рассмотрим, почему меньшие значения ядра снижают переобучение.

Визуализируем эффект, производимый регуляризацией на значения весов

Поскольку работа L2-регуляризатора основана на поощрении меньших значений ядра скрытого плотного слоя, следует ожидать, что значения ядра после обучения меньше в регуляризованном MLP, чем в нерегуляризованном. Как убедиться в этом в TensorFlow.js? Благодаря функции `tfvis.show.layer()` из tfjs-vis можно визуализировать весовые коэффициенты модели TensorFlow.js с помощью одной строки кода. Фрагмент кода в листинге 8.4 демонстрирует, как это сделать. Данный код выполняется по завершении обучения модели MLP. Вызов `tfvis.show.layer()` требует двух аргументов: поверхности визира, на которой происходит визуализация, и визуализируемого слоя.

Листинг 8.4. Визуализируем распределение весовых коэффициентов слоев (из файла jena-weather/index.js)

```
function visualizeModelLayers(tab, layers, layerNames) {
  layers.forEach((layer, i) => {
    const surface = tfvis.visor().surface({name: layerNames[i], tab});
    tfvis.show.layer(surface, layer);
  });
}
```

Произведенная этим кодом визуализация показана на рис. 8.5. В блоках А и Б приведены результаты нерегуляризованного и регуляризованного MLP соответственно. В каждом из этих блоков функция `tfvis.show.layer()` отображает таблицу весов модели со всеми подробностями относительно их названий, формы и количества параметров, минимального/максимального значений параметров и количества нулевых и равных NaN значений параметров (последние бывают полезны для выявления проблемных случаев обучения). Визуализация слоя также содержит кнопки Show Values Distribution (Показать распределение значений) для каждого из весовых коэффициентов модели, при нажатии которых создается гистограмма значений, содержащихся в этом весовом коэффициенте.

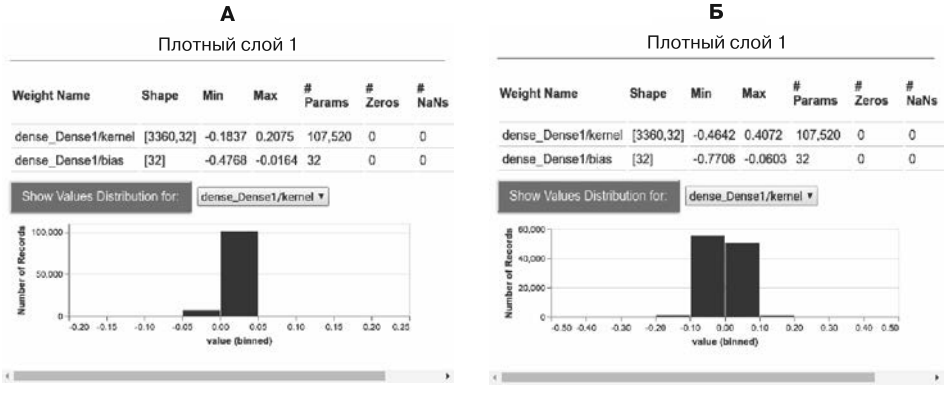


Рис. 8.5. Распределение значений в ядре с L2-регуляризацией (блок А) и без нее (блок Б). Данная визуализация создана с помощью функции tfvis.show.layer(). Обратите внимание на различные масштабы осей X двух гистограмм

При сравнении графиков для двух этих вариантов MLP можно заметить четкое различие: значения ядра распределены по значительно более узкому диапазону при L2-регуляризации. Это заметно как по минимальному/максимальному значениям (первая строка), так и по гистограмме значений. Регуляризация в действии!

Но почему меньшие значения ядра снижают переобучение и улучшают обобщение модели на новые примеры данных? С интуитивной точки зрения можно понять это так, что L2-регуляризация воплощает принцип бритвы Оккама. Вообще говоря, бóльшие величины параметров весов обычно позволяют подогнать модель к нюансам наблюдаемых ею обучающих признаков, а при меньших их величинах модель игнорирует такие нюансы. В предельном случае нулевое значение ядра означает, что модель вообще не учитывает соответствующего входного признака. L2-регуляризация поощряет «экономность» модели, избегая больших величин значений весовых коэффициентов, за исключением случаев, когда это оправданно (когда уменьшение члена «цель — предсказание» перевешивает потери регуляризатора).

L2-регуляризация — лишь одно из множества средств борьбы с переобучением из арсенала специалистов по машинному обучению. В главе 4 мы продемонстрировали возможности слоев дропаута. Это чрезвычайно мощное универсальное средство от

переобучения. Он подходит и для борьбы с переобучением в данной задаче предсказания температуры. Можете убедиться в этом сами, выбрав тип модели MLP with Dropout (MLP с дропаутом) в UI демонстрации. Качество обучения модели при MLP с дропаутом сравнимо с L2-регуляризованным MLP. В подразделе 4.3.2 мы уже обсуждали, как и почему дропаут работает, так что не станем повторяться. Приведем, однако, в табл. 8.1 краткий обзор чаще всего используемых средств для борьбы с переобучением. Он включает интуитивно понятное описание работы каждого из них и соответствующего API TensorFlow.js. Выбрать средство для конкретной задачи можно: 1) следуя примеру хорошо зарекомендовавших себя моделей, применяемых для решения аналогичных задач; 2) рассматривая его как гиперпараметр, с поиском нужного путем оптимизации гиперпараметров (см. подраздел 3.1.2). Кроме того, у каждого из методов снижения переобучения есть настраиваемые параметры, подбираемые в процессе оптимизации гиперпараметров (см. последний столбец табл. 8.1).

Таблица 8.1. Обзор широко используемых методов снижения переобучения в TensorFlow.js

Название метода	**Описание работы метода**	**Соответствующий API TensorFlow.js**	**Основной (-ые) свободный (-е) параметр (-ы)**
L2-регуляризатор	Весовому коэффициенту придается положительное значение потерь (штраф) путем вычисления суммы квадратов значений. Таким образом поощряются меньшие значения параметров	`tf.regularizers.l2()` См., например, подраздел 8.2.3	Коэффициент L2-регуляризации
L1-регуляризатор	Подобно L2-регуляризаторам, поощряются меньшие весовые параметры. Однако значение потерь придается весовому коэффициенту на основе *абсолютных значений* параметров, а не суммы квадратов. Подобное определение потерь для регуляризации приводит к большему числу нулевых параметров весов (то есть «большей разреженности весов»)	`tf.regularizers.l1()`	Коэффициент L1-регуляризации
Сочетание L1- и L2-регуляризаторов	Взвешенная сумма потерь L1- и L2-регуляризации	`tf.regularizers.l1l2()`	Коэффициент L1-регуляризации, коэффициент L2-регуляризации

Продолжение ⇩

Таблица 8.1 *(продолжение)*

Название метода	Описание работы метода	Соответствующий API TensorFlow.js	Основной (-ые) свободный (-е) параметр (-ы)
Дропаут	Выбираемая случайным образом часть входного сигнала обнуляется во время обучения (но не во время выполнения вывода на основе модели), чтобы разорвать ложные корреляции («сговоры», в терминологии Джеффа Хинтона) между весовыми параметрами, возникающие во время обучения	`tf.layers.dropout()` См., например, подраздел 4.3.2	Коэффициент дропаута
Нормализация по батчам	Во время обучения вычисляются среднее значение и среднеквадратичное отклонение входных значений и на основе этих сводных показателей входные данные нормализуются до нулевого среднего значения и единичного среднеквадратичного отклонения	`tf.layers.batchNormalization()`	Различные (см. https://js.tensorflow.org/api/latest/#layers.batchNormalization)
Ранний останов обучения на основе потерь на проверочном наборе данных	Обучение модели сразу прерывается, если значение функции потерь на конец эпохи перестает понижаться	`tf.callbacks.earlyStopping()`	minDelta: пороговое значение, до которого изменения игнорируются. patience: максимально допустимое количество эпох без улучшения ситуации

В завершение этого раздела, посвященного визуализации недообучения и переобучения, приведем простую схему, иллюстрирующую эмпирический способ обнаружения подобных состояний модели (рис. 8.6). Как демонстрирует блок А, при переобучении модель достигает неоптимального (высокого) значения потерь, как на обучающем, так и на проверочном наборе данных. В блоке Б показан типичный паттерн переобучения, когда потери на обучающем наборе данных представляются вполне удовлетворительными (низкими), но потери на проверочном наборе

относительно них не так хороши (выше). Потери на проверочном наборе данных могут останавливаться на каком-то уровне или даже снова расти, хотя потери на обучающем наборе продолжают падать. Желательное для нас состояние модели изображено в блоке В: значения потерь обучающего и проверочного наборов данных не слишком расходятся, так что итоговые потери на проверочном наборе достаточно низки. Учтите, что фраза «достаточно низки» относительна, особенно в тех задачах, для которых ни одна из существующих моделей не может предложить идеального решения. В будущем могут появиться новые модели, которые позволят достичь более низких, по сравнению с показанными в блоке Б, значений потерь. В этом случае приведенный в блоке В паттерн станет случаем недообучения и нам придется взять на вооружение новый тип модели, чтобы от него избавиться, для чего, возможно, снова понадобится пройти через цикл переобучения и регуляризации.

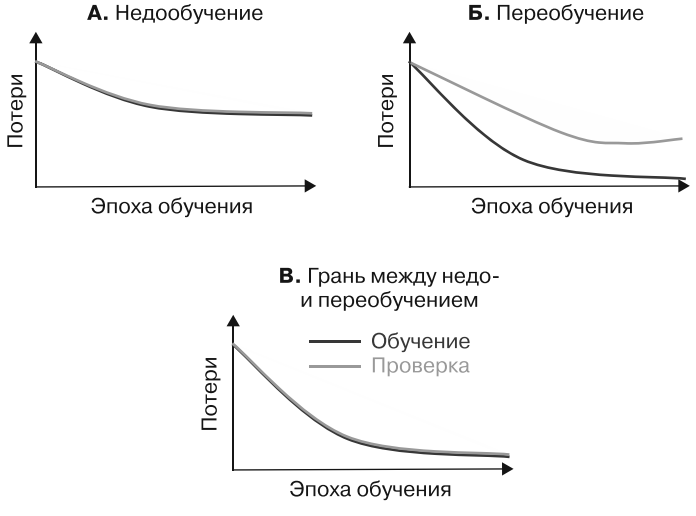


Рис. 8.6. Схематичное изображение кривых потерь для упрощенных случаев недообучения (блок А), переобучения (блок Б) и идеальной подгонки (блок В) модели

Наконец, отметим, что возможности визуализации обучения не ограничиваются функциями потерь. Чтобы можно было облегчить мониторинг процесса обучения, часто визуализируются и другие метрики. Примеры рассыпаны по всей книге. Например, в главе 3 мы строили график кривых ROC, когда обучали бинарный классификатор для обнаружения фишинговых сайтов. Визуализировали мы и матрицу различий, когда обучали классификатор ирисов. В главе 9 мы приведем пример отображения текста, сгенерированного машиной во время обучения генератора текста. Этот пример, хотя и без GUI, предоставляет полезную информацию в режиме реального времени о ходе обучения модели. А именно, глядя на сгенерированный моделью текст, можно получить некое представление о том, насколько хорошо модель в настоящий момент справляется с задачей.

8.3. Универсальный технологический процесс машинного обучения

На текущий момент вы уже познакомились со всеми важными этапами проектирования и обучения моделей МО, включая получение, форматирование, визуализацию и ввод данных, узнали, как выбрать подходящую для набора данных топологию модели и функции потерь, а также рассмотрели само обучение модели. Вы также уже видели некоторые важнейшие типы некорректной работы модели, порой возникающие во время обучения: переобучение и недообучение. Так что самое время окинуть взглядом все изученное и задуматься, какие процессы машинного обучения моделей — общие для различных наборов данных. Полученная в результате абстракция называется *универсальным технологическим процессом машинного обучения* (the universal workflow of machine learning). Мы перечислим по очереди составляющие этого технологического процесса и подробнее обсудим ключевые соображения для каждого из шагов.

1. *Выяснить, является ли машинное обучение правильным подходом.* Во-первых, необходимо обдумать, хорошо ли подходит машинное обучение для решения поставленной задачи, и переходить к следующим шагам только в том случае, если ответ на этот вопрос положительный. В некоторых случаях не основанный на машинном обучении подход даст ничуть не худшие, а то и лучшие результаты при меньших затратах. Например, потратив достаточно усилий на настройку модели, можно научить нейронную сеть «предсказывать» сумму двух целых чисел по этим числам в виде текстовых входных данных (см. пример addition-rnn в репозитории tfjs-examples). Но это далеко не самое эффективное или надежное решение данной задачи: вполне достаточно старой доброй операции сложения на CPU.
2. *Сформулировать задачу машинного обучения и определить, что мы пытаемся предсказать на основе имеющихся данных.* На этом шаге необходимо ответить на два вопроса.
   - *Данные какого вида у нас есть?* При обучении с учителем можно научиться что-либо предсказывать только при наличии маркированных обучающих данных. Например, описанная ранее в главе модель предсказания погоды возможна лишь благодаря наличию набора данных Jena-weather. На этом этапе ограничивающим фактором обычно является доступность данных. Если же доступных данных недостаточно, необходимо собрать еще или, скажем, нанять людей для маркирования вручную немаркированного набора данных.
   - *О задаче какого типа идет речь?* Бинарной классификации, многоклассовой классификации, регрессии или чем-то еще? Именно тип задачи определяет выбор архитектуры модели, функции потерь и т. д.

   Перейти к следующему шагу нельзя, пока мы не определимся с входными и выходными сигналами, а также используемыми данными. Необходимо отдавать себе отчет в неявных гипотезах, принимаемых на этом шаге.

- Гипотеза о возможности предсказания выходных сигналов на основе входных (входной сигнал сам по себе содержит достаточно информации, чтобы модель могла предсказать выходной сигнал для всех возможных примеров данных в конкретной задаче).
- Гипотеза о том, что модели хватит имеющихся данных для усвоения вышеупомянутой зависимости входных и выходных сигналов.

Пока у нас нет работающей модели, это всего лишь гипотезы, которые необходимо проверить или опровергнуть. Не все задачи в принципе решаемы: наличие большого маркированного набора данных соответствий между X и Y не означает, что X содержит достаточно информации для значения Y. Например, попытка предсказания будущей стоимости конкретных акций на основе истории изменения их стоимости, скорее всего, завершится неудачей, поскольку история их цен не содержит достаточно прогнозирующей информации об их будущей цене.

Один из классов нерешаемых задач, о которых вам следует знать: *нестационарные* (nonstationary) задачи, где зависимость входных и выходных данных меняется с течением времени. Представьте себе, что вы пытаетесь создать рекомендательную систему для торговли одеждой (на основе истории покупок конкретного пользователя) и обучаете модель на данных, собранных всего за один год. Основная проблема здесь заключается в том, что вкусы людей к одежде со временем меняются. Модель, достаточно безошибочно работающая на проверочных данных за прошлый год, вовсе не обязательно будет столь же безошибочно работать в этом году. Учтите, что машинное обучение позволяет усваивать лишь те паттерны, которые присутствуют в обучающих данных. В этом случае вполне работоспособным решением будет использование актуальных данных и непрерывное обучение новых моделей.

3. *Найти надежную меру успешности работы обученной модели относительно цели.* В простых задачах такими мерами могут служить безошибочность предсказаний, точность и полнота, кривая ROC и значение AUC (см. главу 3). Но во многих случаях потребуются более сложные предметно-ориентированные метрики, например коэффициент удержания клиентов, — те, что лучше соответствуют более высокоуровневым целям, скажем, цели успешности бизнеса.
4. *Подготовить процесс оценки.* Необходимо спроектировать процесс проверки качества работы модели. В частности, следует разбить данные на три однородных, но непересекающихся множества: обучающий, проверочный и контрольный наборы. Важно, чтобы метки проверочного и контрольного наборов не попали в обучающие данные. Например, в случае временны*х* прогнозов проверочные и контрольные данные должны браться из промежутков времени, *следующих за* обучающими данными. Код предварительной обработки данных должен полностью охватываться тестами для защиты от подобных ошибок.
5. *Выполнить векторизацию данных.* Данные необходимо преобразовать в тензоры, называемые также *n*-мерными массивами, — лингва франка моделей машинного обучения в таких фреймворках, как TensorFlow.js и TensorFlow. Обратите внимание на следующие рекомендации по векторизации данных.

- Числовые значения элементов тензоров обычно нужно масштабировать до небольших и центрированных значений, например находящихся в интервале `[-1, 1]` или `[0, 1]`.
- Если различные признаки (такие как температура и скорость ветра) принимают значения из различных диапазонов (неоднородные данные), то данные необходимо нормализовать. Обычно проводят z-нормализацию к данным с нулевым средним значением и единичным среднеквадратичным отклонением каждого из признаков.

После подготовки тензоров входных и целевых (выходных) данных можно приступать в разработке моделей.

6. *Разработать модель, работающую лучше, чем эталонный подход, основанный просто на здравом смысле*. Разработка модели, которая станет работать лучше эталонной версии, не основанной на машинном обучении (наподобие предсказания средних по населению показателей для задачи регрессии или задача предсказания последней точки данных в ряде данных), позволит продемонстрировать, что машинное обучение действительно может принести вам пользу. Что не всегда так (см. пункт 1).

   Если все нормально, для создания нашей первой модели машинного обучения, работающей лучше эталонной версии, необходимо выбрать три важнейших элемента.

   - *Функция активации последнего слоя* обеспечивает полезные ограничения выходного сигнала модели. Должна соответствовать типу решаемой задачи. Например, в классификаторе фишинговых сайтов из главы 3 для последнего (выходного) слоя использовалась сигма-функция активации, поскольку задача, по своей сути, представляла собой задачу бинарной классификации, а модели предсказания температуры из этой главы использовали линейную функцию активации по причине регрессионной сущности задачи.
   - *Функция потерь*, как и функция активации последнего слоя, должна соответствовать решаемой задаче. Например, для задач бинарной классификации следует использовать `binaryCrossentropy`, для задач многоклассовой классификации — `categoricalCrossentropy`, а для задач регрессии — `meanSquaredError`.
   - *Настройки оптимизатора*. Именно оптимизатор определяет, какими будут обновления весовых коэффициентов нейронной сети. Какой оптимизатор использовать? Какую скорость обучения для него выбрать? Ответы на эти вопросы обычно дает настройка гиперпараметров. Но в большинстве случаев можно спокойно начинать с оптимизатора `rmsprop` и его скорости обучения по умолчанию.

7. *Разработать модель с достаточными разрешающими возможностями, причем чрезмерно подогнанную к обучающим данным*. Необходимо постепенно масштабировать вверх архитектуру своей модели, вручную меняя гиперпараметры, и в итоге получить модель, чрезмерно подогнанную к обучающему набору данных. Как вы помните, основная сложность в обучении с учителем — выбор баланса между *оптимизацией* (подгонкой к данным, которые модель видит во время обучения) и *обобщением* (способностью к безошибочным предсказаниям для не виденных

моделью данных). Идеальная модель балансирует на грани между недообучением и переобучением, то есть между недостаточными и чрезмерными разрешающими возможностями. Чтобы найти эту грань, необходимо сначала ее пересечь.

А чтобы пересечь ее, следует разработать переобученную модель. Обычно это несложно. Можно:

- добавить дополнительные слои;
- сделать уже существующие слои больше;
- увеличить количество эпох обучения модели.

Всегда применяйте визуализацию для мониторинга потерь на обучающем и проверочном наборах данных, а также любых дополнительных интересующих вас метрик (например, AUC) на обоих этих наборах. Если безошибочность модели на проверочном наборе начала падать (см. рис. 8.6, блок Б), значит, модель переобучена.

8. *Добавить в модель регуляризацию и подобрать гиперпараметры.* Следующий шаг — добавление в модель регуляризации и дальнейшая настройка гиперпараметров (обычно автоматически) для максимального приближения к идеальной модели, не недообученной, но и не переобученной. Этот шаг занимает больше всего времени, хотя его можно автоматизировать. Здесь модель многократно модифицируется, обучается, оценивается качество ее работы на проверочном (пока что не контрольном) наборе данных, снова модифицируется, и все это повторяется до тех пор, пока модель не окажется столь хорошей, как только возможно. Что касается регуляризации, имеет смысл попробовать следующее.
   - Добавить слои дропаута с различными коэффициентами дропаута.
   - L1- и/или L2-регуляризацию.
   - Различные архитектуры: добавить или убрать небольшое число слоев.
   - Поменять прочие гиперпараметры (например, количество нейронов плотного слоя).

   Учтите вероятность переобучения на проверочном наборе данных при настройке гиперпараметров. Поскольку гиперпараметры определяются на основе того, насколько хорошо модель работает на проверочном наборе данных, их значения могут слишком хорошо подходить для проверочного набора, а потому плохо обобщаться на прочие данные. Получение несмещенной оценки степени безошибочности модели после настройки гиперпараметров — задача контрольного набора данных. Поэтому не стоит использовать контрольный набор данных во время настройки гиперпараметров.

Это универсальный технологический процесс машинного обучения! В главе 12 мы добавим в него еще два шага, ориентированных на практическое использование (шаг оценки и шаг развертывания). Но пока это готовый рецепт перехода от расплывчатой идеи машинного обучения к обученной и готовой выдавать полезные предсказания модели.

С этим фундаментом знаний далее мы приступим к изучению более продвинутых типов нейронных сетей. И начнем в главе 9 с моделей, предназначенных для последовательных данных.

Упражнения

1. В задаче предсказания температуры мы обнаружили, что линейный регрессор существенно недообучается на данных и выдает плохие предсказания как на обучающем, так и на проверочном наборе данных. Может ли добавление L2-регуляризации улучшить степень безошибочности подобной недообученной модели? Проверьте сами, это несложно, для чего модифицируйте функцию `buildLinearRegressionModel()` в файле `jena-weather/models.js`.
2. При предсказании завтрашней температуры в примере Jena-weather для генерации входных признаков использовались данные за десять предыдущих дней. Возникает естественный вопрос: что, если использовать более длительный промежуток времени? Повысится ли степень безошибочности предсказаний при включении большего количества данных? Узнать это вы можете, модифицировав `const lookBack` в файле `jena-weather/index.js` и запустив обучение в браузере (например, воспользовавшись MLP с L2-регуляризацией). Конечно, более длительный интервал времени приведет к увеличению размера входных признаков и удлинению обучения. Так что обратная сторона этого вопроса: можно ли использовать более короткий промежуток времени без существенного ущерба для безошибочности предсказаний? Попробуйте и это.

Резюме

- Модуль tfjs-vis может помочь с визуализацией процесса обучения модели МО в браузере. Если точнее, мы показали, как использовать tfjs-vis:
  - для визуализации топологии моделей TensorFlow.js;
  - построения графиков кривых потерь и метрик во время обучения;
  - вывода распределения значений весовых коэффициентов после обучения.
- Мы продемонстрировали конкретные примеры этих технологических процессов визуализации.
- Недообучение и переобучение — основополагающие виды поведения моделей машинного обучения, требующие мониторинга и должного понимания в каждой задаче машинного обучения. Обнаружить их можно путем сравнения кривых потерь на обучающем и проверочном наборах данных во время обучения. С помощью встроенного метода `tfvis.show.fitCallbacks()` можно легко визуализировать эти кривые в браузере.
- Универсальный технологический процесс машинного обучения — список характерных шагов и рекомендуемых практик различных типов задач обучения с учителем. Он простирается от выяснения сущности задачи и требований к данным до поиска модели точно на грани между недообучением и переобучением.

9 Глубокое обучение для последовательностей и текста

В этой главе

- Чем последовательные данные отличаются от непоследовательных.
- Какие методики глубокого обучения подходят для задач с последовательными данными.
- Представления текста в глубоком обучении, включая унитарное кодирование, федеративное кодирование и вложения слов.
- Что такое RNN и почему они хорошо подходят для задач с последовательными данными.
- Что такое одномерная свертка и почему это удачная альтернатива RNN.
- Уникальные особенности задач преобразования последовательностей в последовательности и использование для их реализации механизма внимания.

В этой главе мы сосредоточимся на задачах, в которых участвуют последовательные данные. Главное в таких данных — упорядоченность элементов. Как вы, наверное, догадались, мы уже имели дело с последовательными данными. А именно, последовательными являются данные Jena-weather, с которыми вы познакомились в главе 7. Их можно представить в виде массива массивов чисел. Упорядоченность, безусловно, важна для внешнего массива, поскольку измерения поступают в определенные моменты времени. Если поменять порядок элементов во внешнем массиве на

противоположный, например тенденцию к росту атмосферного давления превратить в тенденцию к убыванию, прогноз погоды получится совершенно иным. Последовательные данные встречаются повсеместно: курсы акций, электрокардиограммы (ЭКГ), строки символов в коде программного обеспечения, последовательные кадры видео и последовательности действий робота. Сравните эти примеры с непоследовательными данными наподобие набора данных iris-flowers из главы 3: изменение порядка четырех числовых признаков (длина и ширина чашелистиков и лепестков) в них ни на что не повлияет[1].

В первой части главы мы познакомим вас с упоминавшимся в главе 1 замечательным типом моделей — RNN, рекуррентными нейронными сетями, специально предназначенными для усвоения паттернов из последовательных данных. Мы научимся интуитивно понимать, благодаря каким особым признакам RNN оказываются чувствительны к упорядоченности элементов и содержащейся в ней информации.

Во второй части главы займемся, вероятно, чаще всего встречающимся (особенно в Интернете!) частным случаем последовательных данных — текстом. Начнем с представления текста в глубоком обучении и применения RNN к подобным представлениям, а затем перейдем к одномерным сверточным сетям и поговорим об их широких возможностях по обработке текста и о том, почему их имеет смысл использовать вместо RNN для определенных типов задач.

В последней части главы мы пойдем еще дальше и изучим связанные с последовательностями задачи, несколько более сложные, чем представление числа или класса. В частности, отважимся на обсуждение задач преобразования последовательностей в последовательности — предсказания выходной последовательности по входной. И проиллюстрируем на примере решение простейшей задачи преобразования последовательностей в последовательности с помощью новой архитектуры модели — *механизма внимания* (attention mechanism), которая играет все более и более важную роль в сфере обработки текстов на естественном языке с помощью глубокого обучения.

К концу данной главы вы изучите основные типы последовательных данных в глубоком обучении, их представления в виде тензоров и научитесь пользоваться TensorFlow.js для создания простых RNN, одномерных сверточных сетей и сетей внимания для решения задач машинного обучения, связанных с последовательными данными.

Слои и модели из этой главы — одни из самых сложных в книге. Это цена, которую приходится платить за их повышенные разрешающие возможности в задачах обучения с последовательными данными. Вероятно, вам непросто будет с первого раза разобраться в некоторых из них, хотя мы и приложили максимум усилий, чтобы описать их как можно более интуитивно понятным образом, с помощью схем и псевдокода. В случае затруднений попробуйте поэкспериментировать с примерами кода и выполнить упражнения, размещенные в конце главы. По нашему опыту, на практике сложные концепции и архитектуры, подобные встречающимся в этой главе, усваиваются намного проще.

---

[1] Убедитесь в этом, решив упражнение 1 в конце главы.

9.1. Вторая попытка прогноза погоды: знакомство с RNN

Модели, созданные нами для задачи Jena-weather в главе 8, игнорировали информацию о порядке элементов. В этом разделе мы объясним, почему это важно и как учесть эту информацию с помощью RNN. Благодаря этому сможем в задаче предсказания температуры добиться более высокой степени безошибочности.

9.1.1. Почему плотные слои не способны моделировать упорядоченность

Поскольку в предыдущей главе мы уже подробно описывали набор данных Jena-weather, здесь разговор об этом наборе данных и соответствующей задаче машинного обучения будет коротким. Задача состоит в предсказании температуры через 24 часа после определенного момента времени на основе показаний 14 метеорологических приборов (температура, атмосферное давление, скорость ветра и т. д.) за десять предшествующих этому моменту дней. Показания приборов сняты с интервалом десять минут, но мы субдискретизируем их в шесть раз, до одного показания в час, ради уменьшения размера модели и длительности обучения. Таким образом, форма тензора признаков каждого обучающего примера данных — `[240, 14]`, где `240` — число интервалов измерений за десять дней, а `14` — количество показаний приборов.

Пытаясь решить эту задачу с помощью моделей линейной регрессии и MLP в предыдущей главе, мы схлопывали двумерные входные признаки до одномерных с помощью слоя `tf.layers.flatten` (см. листинг 8.2 и рис. 8.2). Шаг схлопывания нужен, поскольку и в линейном регрессоре, и в MLP для обработки входных данных используются плотные слои, для которых входные данные должны быть одномерными (для отдельного входного примера). Поэтому информация о всех временны*х* шагах перемешивается таким образом, что перестает иметь значение, какой шаг первый, а какой — следующий за ним, какой шаг следует за каким, насколько удалены друг от друга два шага и т. д. Иными словами, при схлопывании двумерного тензора формы `[240, 14]` в одномерный тензор формы `[3360]` порядок этих 240 шагов перестает играть роль, главное, чтобы на этапах обучения и вывода все было одинаково. Убедиться в этом экспериментально вы можете, решая упражнение 1, размещенное в конце данной главы. А с теоретической точки зрения подобную нечувствительность к порядку элементов данных можно представить следующим образом. В основе плотного слоя лежит набор линейных уравнений, в каждом из которых каждый входной признак $[x_1, x_2 ... x_n]$ умножается на настраиваемый коэффициент из ядра $[k_1, k_2 ... k_n]$:

$$y = f(k_1x_1 + k_2x_2 + ... + k_nx_n). \quad (9.1)$$

На рис. 9.1 приведена наглядная схема работы плотного слоя: ведущие от входных элементов к выходу слоя пути визуально симметричны друг другу, отражая

математическую симметрию уравнения 9.1. При работе с последовательными данными симметрия *нежелательна*, поскольку при этом визуализация игнорирует порядок элементов.

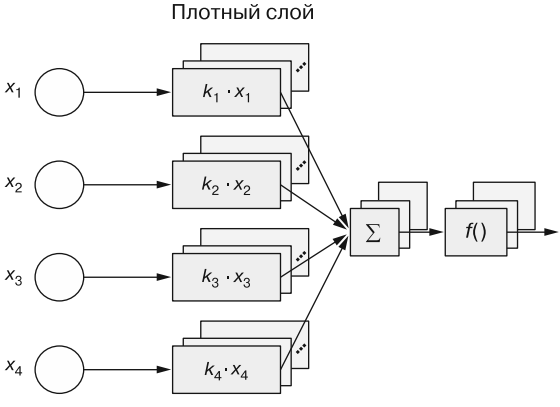


Рис. 9.1. Внутренняя архитектура плотного слоя. Плотный слой выполняет умножение и сложение симметрично относительно входных данных, в отличие от слоя simpleRNN (рис. 9.2), где симметрия нарушается благодаря пошаговым вычислениям. Мы предполагаем, что входной сигнал состоит всего лишь из четырех элементов, и опускаем члены смещения ради простоты. Кроме того, здесь показаны операции только для одного выходного нейрона плотного слоя. Остальные нейроны показаны в виде стопки затемненных прямоугольников

Существует простой способ показать, что подход на основе плотных слоев (многослойных перцептронов, даже с регуляризацией) — не слишком хорошее решение задачи предсказания температуры. Для этого нужно сравнить степень его безошибочности с аналогичным показателем подхода без машинного обучения, на основе одного здравого смысла.

Что такое подход на основе здравого смысла? Он состоит в предсказании значения температуры, равного последнему показанию температуры из входных признаков. Проще говоря, считаем, что температура через 24 часа будет такой же, как сейчас! Этот подход интуитивно понятен, ведь из каждодневного опыта мы знаем: завтрашняя температура обычно близка к сегодняшней (точно в то же время дня, конечно). Это очень простой алгоритм, дающий неплохие результаты, лучшие, чем все прочие столь же простые алгоритмы (например, применение в качестве предсказания температуры значения двухдневной давности).

Каталог `jena-weather` репозитория tfjs-examples, который мы использовали в главе 8, содержит команду для оценки безошибочности подхода на основе здравого смысла:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/jena-weather
yarn
yarn train-rnn --modelType baseline
```

Команда `yarn train-rnn` вызывает сценарий `train-rnn.js` и выполняет вычисления в серверной среде на основе Node.js[1]. Мы вернемся к этому режиму операций чуть позже, при обсуждении RNN. В результате выполнения этой команды на экран будет выведено нечто вроде:

```
Commonsense baseline mean absolute error: 0.290331
```

Таким образом, при подходе на основе здравого смысла, без применения машинного обучения, получаем среднюю абсолютную погрешность предсказания около 0,29 (в нормализованном виде), что практически идентично (возможно, даже чуть лучше) наилучшим показателям погрешности проверки модели MLP в главе 8 (см. рис. 8.4). Другими словами, MLP, с и без регуляризации, не способен превзойти показатели эталонного подхода на основе здравого смысла!

Подобные наблюдения не редкость в машинном обучении: не так-то просто машинному обучению превзойти подход на основе здравого смысла. Иногда для этого приходится долго и тщательно проектировать модель машинного обучения или подбирать гиперпараметры. Упомянутое наблюдение также подчеркивает важность создания не основанного на машинном обучении эталона для сравнения в ходе работы с задачами машинного обучения. Безусловно, не хотелось бы тратить усилия на создание алгоритма машинного обучения, не способного даже превзойти намного более простой и менее затратный в вычислительном отношении эталонный алгоритм! Сможем ли мы превзойти эталонный алгоритм в задаче предсказания температуры? Да, сможем, и в этом нам помогут RNN. Давайте взглянем, как RNN захватывают и анализируют порядок последовательных данных.

9.1.2. Моделирование последовательного упорядочения с помощью RNN

Блок A на рис. 9.2 демонстрирует внутреннюю структуру слоя RNN на короткой последовательности из четырех элементов. Существует несколько вариантов слоев RNN, из которых на схеме показан простейший, который называется simpleRNN и доступен в TensorFlow.js в виде фабричной функции `tf.layers.simpleRNN()`. Далее в этой главе мы обсудим и более сложные варианты RNN, но пока что сосредоточимся на simpleRNN.

Эта схема демонстрирует пошаговую обработку временных срезов входных признаков (x_1, x_2, x_3...). На каждом шаге функция $f()$, представленная прямоугольником в центре схемы, обрабатывает x_i и выдает выходной сигнал y_i, из которого в сочетании со следующим входным срезом x_{i+1} получается входной сигнал — функция

[1] Код, реализующий подход на основе здравого смысла, без машинного обучения, вы найдете в функции getBaselineMeanAbsoluteError() в файле jena-weather/models.js. Для обхода всех батчей проверочного поднабора данных, вычисления потерь MAE для каждого из них и накопления всех значений потерь для получения итоговых потерь в нем используется метод forEachAsync() объекта Dataset.

$f()$ — для следующего шага. Важно отметить: хотя в схеме показаны четыре отдельных прямоугольника с описаниями функций в них, на самом деле они представляют одну и ту же функцию. Функция $f()$ называется *ячейкой* (cell) RNN-слоя и используется в цикле во время его работы. Следовательно, RNN-слой можно считать RNN-ячейкой, обернутой в цикл `for`[1].

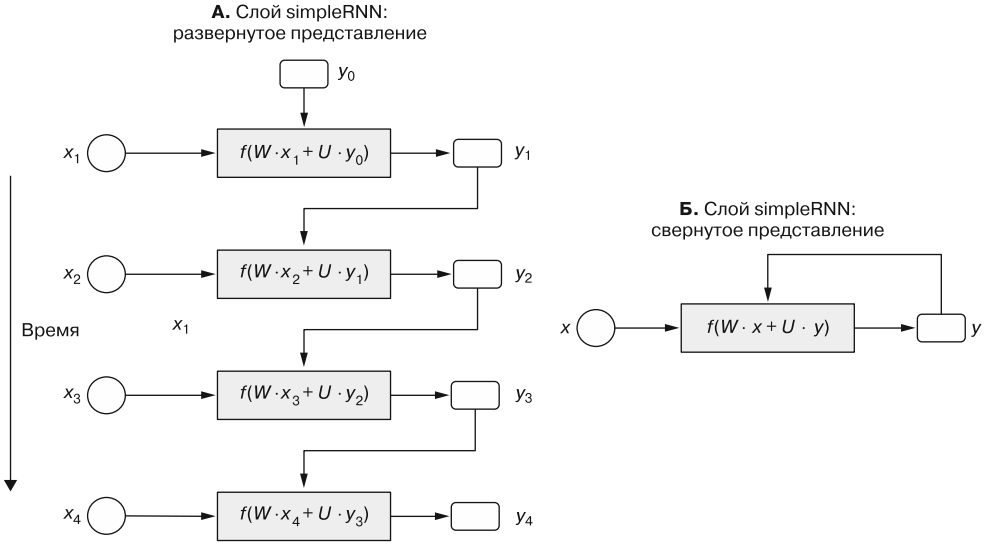


Рис. 9.2. Развернутое (блок А) и свернутое (блок Б) представление внутренней структуры simpleRNN. Свернутое представление отражает тот же алгоритм, что и развернутое, только лаконичнее, иллюстрируя последовательную обработку входных данных simpleRNN более кратко. В блоке Б показано соединение, идущее обратно от выходного сигнала (y) в саму модель, из-за которого такие слои называются рекуррентными (recurrent). Как и на рис. 9.1, отображены лишь четыре входных элемента, а члены смещений ради простоты опущены

Если сравнить структуры simpleRNN и плотного слоя (см. рис. 9.1), можно увидеть два основных различия.

- SimpleRNN обрабатывает входные элементы (временны*е* шаги) по одному, отражая последовательную природу входных данных, на что плотный слой не способен.
- В simpleRNN в результате обработки на каждом входном временно́м шаге генерируется выходной сигнал y_i. Выходной сигнал с предыдущего временно́го шага (например, y_1) используется слоем при обработке следующего временно́го шага (скажем, x_2). Именно поэтому в название RNN (recurrent neural network — рекуррентная нейронная сеть) входит слово *«рекуррентный»*: выходной сигнал с предыдущих временных шагов передается назад и становится входным сигналом для последующих временных шагов. В таких типах слоев, как плотный слой,

[1] Высказывание, приписываемое Юджину Бревдо.

conv2d и maxPooling2d, рекуррентности нет, выходная информация не передается в них обратно, поэтому они получили название слоев *прямого распространения* (feedforward layers).

Благодаря этим уникальным особенностям simpleRNN разрушает симметрию между входными элементами и чувствителен к упорядоченности входных элементов. Если переупорядочить элементы последовательных входных данных, выходные данные также изменятся. Это отличает simpleRNN от плотного слоя.

В блоке Б на рис. 9.2 приведено более абстрактное представление simpleRNN. Его называют *свернутой* (rolled) диаграммой RNN, в отличие от развернутой (unrolled) диаграммы в блоке А, поскольку все временные шаги в ней свернуты в один цикл. Свернутая диаграмма соответствует циклу `for` в языках программирования, с помощью которого simpleRNN и прочие типы RNN реализуются внутри TensorFlow.js. Но вместо того, чтобы демонстрировать сам код, давайте взглянем на приведенный в листинге 9.1 намного более краткий псевдокод simpleRNN — реализацию архитектуры simpleRNN, приведенной на рис. 9.2. Он позволит вам сосредоточиться на самой сути работы слоя RNN.

Листинг 9.1. Псевдокод внутренних вычислений simpleRNN

```
y = 0
for x in input_sequence:
  y = f(dot(W, x) + dot(U, y))
```

y соответствует y на рис. 9.2. Начальные значения состояния равны нулю

x соответствует x на рис. 9.2. Цикл for обходит все временные шаги входной последовательности

W и U — матрицы весовых коэффициентов входного сигнала и состояния (выходной сигнал возвращается обратно и становится рекуррентным входным сигналом) соответственно. Именно здесь выходной сигнал временного шага i и становится состоянием (рекуррентным входным сигналом) для временного шага i + 1

Из листинга 9.1 видно, как выходной сигнал для временно́го шага `i` становится состоянием для следующего временно́го шага (следующей итерации цикла). *Состояние* (state) — важное понятие для RNN, с его помощью она запоминает, что произошло на уже виденных ею временны*х* шагах входной последовательности. В цикле `for` это состояние памяти объединяется с последующими временны*ми* шагами и становится новым состоянием памяти. Благодаря этому simpleRNN может по-разному реагировать на один и тот же входной элемент в зависимости от предыдущих элементов последовательности. Подобная чувствительность на основе памяти — краеугольный камень последовательной обработки. Простой пример: при декодировании кода Морзе, состоящего из точек и тире, значение тире зависит от последовательности точек и тире перед ним (и после него). Еще один пример: в английском языке слово *last* может означать совершенно разные вещи в зависимости от предшествующих слов.

Название simpleRNN очень удачно, поскольку его выходной сигнал и состояние совпадают. Позднее мы обсудим более сложные и перспективные архитектуры RNN. В некоторых из них выходной сигнал и состояние — две отдельные сущности, а в прочих бывает даже несколько состояний.

Еще стоит отметить, что благодаря циклу `for` RNN могут обрабатывать входные последовательности из произвольного числа шагов. Подобное нельзя осуществить

путем схлопывания последовательных входных данных и подачи их на вход плотного слоя, так как он принимает входной сигнал только фиксированной формы.

Более того, цикл `for` отражает еще одно важное свойство RNN — *единство параметров* (parameter sharing). Под ним понимается использование одних и тех же весовых параметров (`W` и `U`) для всех временны*х* шагов. Можно взять и собственное значение `W` (и `U`) для каждого из временны*х* шагов, но это нежелательно, поскольку ограничивает число временны*х* шагов, которое может обработать RNN, и ведет к резкому росту числа настраиваемых параметров, а значит, и росту объема вычислений и вероятности переобучения. Таким образом, RNN-слои подобны слоям conv2d в сверточных сетях тем, что для эффективности вычислений и защиты от переобучения используется единство параметров, хотя рекуррентные и conv2d-слои реализуют его по-разному. Слои conv2d применяют трансляционную инвариантность относительно пространственных измерений, а RNN-слои — относительно измерения *времени*.

На рис. 9.2 изображено происходящее в simpleRNN во время выполнения вывода (при прямом проходе). На нем не показано обновление весовых параметров (`W` и `U`) во время обучения (обратный проход). Однако обучение RNN следует правилам обратного распространения ошибки, описанным в подразделе 2.2.2 (см. рис. 2.8), — начинается с функции потерь, прохода обратно по списку операций с взятием их производных и накопления по мере этого значений градиентов. Математически обратный проход рекуррентной сети не отличается от прямого. Единственное отличие — при обратном проходе RNN-слоя мы идем обратно во времени по развернутому графу наподобие приведенного в блоке A на рис. 9.2. Именно поэтому процесс обучения RNN иногда называют *обратным распространением во времени* (backpropagation through time, BPTT).

SimpleRNN в действии

Хватит абстрактных рассуждений о simpleRNN и RNN в целом. Давайте посмотрим, как создать слой simpleRNN и включить его в объект модели, чтобы предсказывать с его помощью температуру точнее, чем раньше. В коде из листинга 9.2 (отрывок из файла `jena-weather/train-rnn.js`) показано, как это делается. Несмотря на внутреннюю сложность слоя simpleRNN, сама модель очень проста. Она состоит всего из двух слоев. Первый из них — simpleRNN на 32 нейрона. Второй — плотный слой, в котором для генерации непрерывных численных предсказаний температуры используется линейная функция активации по умолчанию. Отметим, что поскольку модель начинается со слоя RNN, схлопывать последовательные входные данные больше не нужно (сравните с листингом 8.3, где мы создавали MLP для решения той же задачи). На самом деле, если поместить слой схлопывания перед слоем simpleRNN, будет сгенерирована ошибка, поскольку входные сигналы слоев RNN в TensorFlow.js должны быть как минимум трехмерными (включая измерение батчей).

Увидеть эту модель simpleRNN в действии можно с помощью следующей команды:

```
yarn train-rnn --modelType simpleRNN --logDir /tmp/
  jean-weather-simpleRNN-logs
```

Листинг 9.2. Создание модели на основе simpleRNN для задачи предсказания температуры

```
function buildSimpleRNNModel(inputShape) {
  const model = tf.sequential();
  const rnnUnits = 32;
  model.add(tf.layers.simpleRNN({
    units: rnnUnits,
    inputShape
  }));
  model.add(tf.layers.dense({units: 1}));
  return model;
}
```

Это зашитое в код количество нейронов слоя simpleRNN, демонстрирующее хорошие результаты, определено путем подбора гиперпараметра вручную

Первый слой модели — слой simpleRNN. Схлопывать последовательный входной сигнал формы [null, 240, 14] не нужно

Наша модель завершается плотным слоем с одним нейроном и линейной функцией активации по умолчанию для задачи регрессии

Обучается эта модель RNN в серверной среде с помощью tfjs-node. Из-за большого объема вычислений, необходимых для основанного на BPTT обучения RNN, обучение этой модели в браузерной среде с ее ограничениями на ресурсы заняло бы больше времени и потребовало бы значительных усилий, а то и вообще оказалось невозможным. При наличии настроенной должным образом среды CUDA можно добавить в команду флаг `--gpu` и значительно ускорить обучение.

Благодаря заданию флага `--logDir` в предыдущей команде процесс обучения модели журналирует значения потерь в указанный каталог. При желании можно затем загрузить данные и построить график кривых потерь в браузере с помощью утилиты TensorBoard. Рисунок 9.3 представляет собой снимок экрана TensorBoard. На уровне JavaScript-кода это реализуется с помощью указания в вызове `tf.LayersModel.fit()` особой функции обратного вызова, указывающей на каталог с журналами. Более подробное описание можно найти в инфобоксе 9.1.

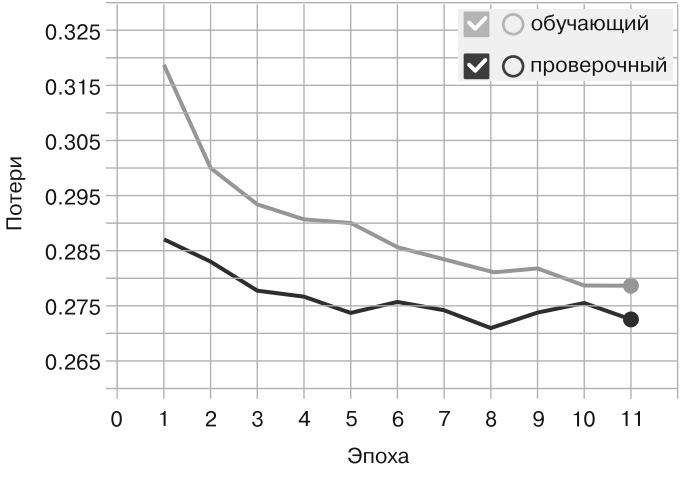


Рис. 9.3. Кривые потерь MAE из созданной для задачи предсказания температуры Jena модели simpleRNN. Эта диаграмма представляет собой снимок экрана TensorBoard, обрабатывающего журналы обучения модели simpleRNN с помощью Node.js

ИНФОБОКС 9.1. Использование обратных вызовов TensorBoard для мониторинга длительного обучения модели в Node.js

В главе 8 мы познакомили вас с функциями обратного вызова из библиотеки tfjs-vis, предназначенными для мониторинга вызовов `tf.LayersModel.fit()` в браузере. Впрочем, tfjs-vis — чисто браузерная библиотека и для работы с Node.js не подходит. По умолчанию `tf.LayersModel.fit()` в tfjs-node (или tfjs-node-gpu) отрисовывает индикаторы хода выполнения и отображает потери и показатели времени в терминале. Несмотря на простоту и информативность, текст и числа часто интуитивно не так понятны и привлекательны в качестве средства мониторинга длительного обучения моделей, как GUI. Например, небольшие изменения значения потерь за продолжительное время, часто как раз и интересующие нас на последних этапах обучения модели, намного проще заметить на графике (при заданных должным образом масштабе и сетке), чем в массиве текста.

К счастью, в серверной среде нам на помощь приходит утилита *TensorBoard*. Она изначально предназначалась для TensorFlow (Python), но tfjs-node и tfjs-node-gpu умеют записывать данные в подходящем для ввода в TensorBoard формате. Для журналирования потерь и значений показателей в TensorBoard из вызовов `tf.LayersModel.fit()` и `tf.LayersModel.fitDataset()` воспользуйтесь следующим шаблоном:

```
import * as tf from '@tensorflow/tfjs-node';
// Или '@tensorflow/tfjs-node-gpu'
    // …
await model.fit(xs, ys, {
  epochs,
  callbacks: tf.node.tensorBoard('/path/to/my/logdir')
});
    // Или для fitDataset():
await model.fitDataset(dataset, {
  epochs,
  batchesPerEpoch,
  callbacks: tf.node.tensorBoard('/path/to/my/logdir')
});
```

В результате этих вызовов значения потерь и все заданные во время вызова `compile()` показатели будут записаны в каталог `/path/to/my/logdir`. Для просмотра этих журналов в браузере сделайте следующее.

1. Откройте отдельное окно терминала.
2. Установите TensorBoard с помощью следующей команды (если он еще не установлен):

   ```
   pip install tensorboard
   ```

3. Запустите сервер прикладной части TensorBoard, указав при этом каталог журналов, заданный при создании обратного вызова:

   ```
   tensorboard –logdir /path/to/my/logdir
   ```

4. Перейдите в браузере на отображаемый процессом TensorBoard URL вида `http://`. После этого в прекрасном веб-интерфейсе TensorBoard появятся диаграммы потерь и показателей наподобие приведенных на рис. 9.3 и 9.5.

Текстовая сводка топологии модели simpleRNN, созданной в листинге 9.2, выглядит следующим образом:

```
Layer (type)                 Output shape              Param #
     =================================================================
     simple_rnn_SimpleRNN1 (Simpl [null,32]                 1504
     _________________________________________________________________
     dense_Dense1 (Dense)         [null,1]                  33
     =================================================================
Total params: 1537
     Trainable params: 1537
     Non-trainable params: 0
     _________________________________________________________________
```

В ней намного меньше весовых параметров, чем в ранее описанном MLP (1537 вместо 107 585, то есть в 70 раз меньше), но потери MAE на проверочном наборе данных ниже (то есть предсказания более точны), чем у MLP во время обучения (0,271 вместо 0,289). Это небольшое, но стабильное сокращение погрешности предсказания температуры подчеркивает мощь единства параметров на основе временно́й инвариантности и преимущества RNN при обучении на последовательных данных, такие как наши метеорологические измерения.

Наверное, вы обратили внимание на то, что, несмотря на относительно небольшое число весовых параметров в слое simpleRNN, его обучение и выполнение вывода происходят намного дольше, чем в моделях прямого распространения, подобных MLP. Этот основной недостаток RNN — следствие невозможности распараллеливания операций над отдельными временны*ы*ми шагами. А невозможно распараллеливание потому, что последующие шаги зависят от значений состояния, вычисляемых на предыдущих шагах (см. рис. 9.2 и псевдокод в листинге 9.1). В нотации «О большое»: прямой проход RNN занимает $O(n)$ времени, где n — число входных временны*ы*х шагов. Обратный проход (BPTT) занимает еще $O(n)$ времени. Входной признак задачи Jena-weather состоит из большого количества (240) временны*ы*х шагов, в результате чего обучение и длится так долго, как мы видели ранее. Именно поэтому мы обучаем модель в tfjs-node, а не в браузере.

Ситуация с RNN разительно отличается от ситуации со слоями прямого распространения, например плотными и conv2d. В этих слоях можно распараллелить вычисления по входным элементам, поскольку операция над одним элементом не зависит от результатов, которые дают другие входные элементы. Благодаря этому прямой и обратный проходы для подобных слоев прямого распространения занимают менее $O(n)$ времени (в некоторых случаях около $O(1)$) при использовании GPU для ускорения вычислений. В разделе 9.2 мы изучим еще несколько распараллеливаемых подходов к моделированию последовательных данных, в частности одномерную свертку. Впрочем, познакомиться с RNN все равно не помешает, ведь одномерная свертка не обладает в определенном смысле такой чувствительностью к позиции элементов последовательности, как они (больше об этом рассказывается далее).

Шлюзовой рекуррентный блок — более сложный тип RNN

SimpleRNN — не единственный рекуррентный слой в TensorFlow.js. Есть еще два: шлюзовой рекуррентный блок (gated recurrent unit, GRU[1]) и LSTM, что, как вы помните, расшифровывается как «сеть с долгой краткосрочной памятью»[2]. В большинстве сценариев, реализуемых на практике, имеет смысл использовать один из них. Слой SimpleRNN слишком прост для большинства практических задач, хотя и требует намного меньшего объема вычислений и его внутренние механизмы проще для понимания, чем GRU и LSTM. У simpleRNN есть одна главная проблема: хотя теоретически на момент t он способен хранить информацию о входных данных, виденных многими временны*ы*ми шагами ранее, усвоение подобных долговременных зависимостей на практике представляет собой непростую задачу.

Причина этого заключается в *проблеме исчезающего градиента* (vanishing gradient problem) — эффекте, аналогичном наблюдаемому в сетях прямого распространения глубиной во много слоев: по мере добавления новых слоев в сеть градиенты, распространяемые обратно от функции потерь в начальные слои, все уменьшаются и уменьшаются. А значит, обновления весовых коэффициентов тоже все уменьшаются и уменьшаются — до такой степени, что сеть постепенно становится необучаемой. В случае RNN роль большого числа слоев в этой проблеме играет большое число временны*х* шагов. GRU и LSTM представляют собой RNN, специально спроектированные для решения проблемы исчезающего градиента, причем из них GRU — более простой. Давайте взглянем, как GRU это реализует.

Внутренняя структура GRU сложнее структуры simpleRNN. На рис. 9.4 показано свернутое представление внутренней структуры GRU. По сравнению с аналогичным свернутым представлением simpleRNN (блок Б на рис. 9.2) оно включает большее число компонентов. Входной сигнал (x) и выходной сигнал/состояние (в литературе по RNN традиционно обозначаемые h) проходят через *четыре* уравнения, превращаясь в новый выходной сигнал/состояние. Сравните это с simpleRNN с *одним-единственным* уравнением. Сложность GRU отражается также в псевдокоде (листинг 9.3), который можно считать реализацией механизмов, показанных на рис. 9.4. Для простоты члены смещения в этом псевдокоде опущены.

Отметим два наиболее важных нюанса внутреннего устройства GRU.

1. Слои GRU сильно упрощают перенос информации на большое число временны*х* шагов. Это достигается за счет промежуточной величины z, называемой *шлюзом обновления* (update gate). Благодаря шлюзу обновления GRU обучается переносить состояние на большое число временны*х* шагов с минимальными изменениями. В частности, в уравнении $(1 - z)h + zh'$ при $z = 0$ состояние h просто копируется с одного временно́го шага на следующий. Возможность подобного переноса в неизменном виде играет важную роль в том, как GRU борется с проб-

[1] *Cho K. et al.* Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. — 2014. https://arxiv.org/abs/1406.1078.

[2] *Hochreiter S., Schmidhuber J.* Long Short-Term Memory // Neural Computation. Vol. 9. 1997. № 8. P. 1735–1780.

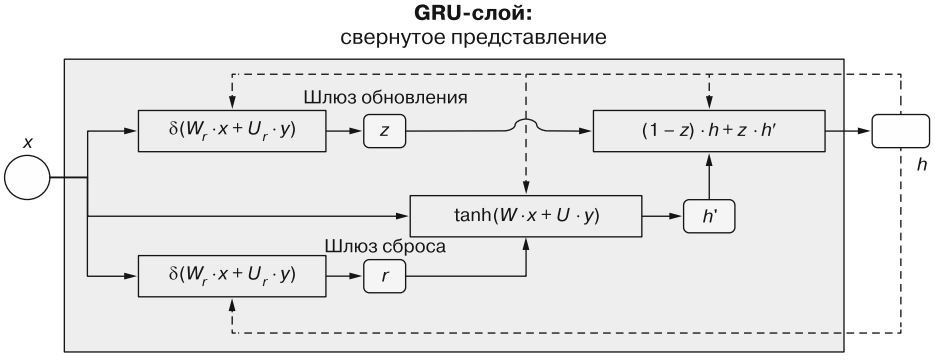


Рис. 9.4. Свернутое представление GRU-ячейки — более сложного и обладающего большими возможностями, чем simpleRNN, типа RNN-слоя. Это свернутое представление, сравнимое с блоком Б на рис. 9.2. Учтите, что в этих уравнениях для простоты мы опустили члены смещения. Пунктирные линии указывают на соединения обратной связи, ведущие от выходного сигнала GRU-ячейки (h) в ту же ячейку на последующих временны́х шагах

Листинг 9.3. Псевдокод GRU-слоя

```
h = 0                                          # h соответствует h на рис. 9.4. Как и в simpleRNN,
                                               # в начале состояние инициализируется нулями
for x_i in input_sequence:                     # Цикл обхода всех временных шагов
                                               # входной последовательности
    z = sigmoid(dot(W_z, x) + dot(U_z, h))     # z называется шлюзом обновления
    r = sigmoid(dot(W_r, x) + dot(W_r, h))     # r называется шлюзом сброса
    h_prime = tanh(dot(W, x) + dot(r, dot(U, h)))  # h_prime — текущее
                                               # временное состояние
    h = dot(1 - z, h) + dot(z, h_prime)        # Новое состояние формируется путем взвешенного
                                               # сочетания (z играет роль весового коэффициента)
                                               # h_prime (текущего временного состояния)
                                               # и h (предыдущего состояния)
```

лемой исчезающего градиента. Шлюз обновления z вычисляется как линейная комбинация входного сигнала x и текущего состояния h с последующим применением нелинейной сигма-функции.

2. Помимо шлюза обновления z, в GRU есть еще один шлюз — так называемый *шлюз сброса* (reset gate) r. Подобно шлюзу обновления z, r вычисляется как нелинейная сигма-функция, применяемая к линейной комбинации входного сигнала и текущего состояния h. Шлюз сброса определяет забываемую часть текущего состояния. В частности, в уравнении $\tanh(Wx + rUh)$ при $r = 0$ текущее состояние h ни на что не влияет, и если $(1 - z)$ в последующем уравнении тоже близко к нулю, то влияние текущего состояния h на следующее состояние будет минимальным. Таким образом, благодаря r и z совместно GRU обучается забывать историю — или ее часть — при соответствующих условиях. Допустим, мы пытаемся классифицировать обзор фильма как позитивный или негативный. Обзор может начинаться с фразы: «Фильм довольно неплох», но где-то дальше

в нем может говориться: «Впрочем, этот фильм хуже других лент, основанных на той же идее». К этому моменту первоначальная похвала уже должна более или менее забыться, поскольку именно продолжение обзора должно обладать бóльшим весом при окончательном определении его тональности.

Вот так, если описывать очень грубо и в общих чертах, работает GRU. Важно помнить, что внутренняя структура GRU дает возможность RNN усваивать, когда следует переносить старое состояние, а когда обновлять состояние информацией из входных сигналов. Это обучение олицетворяется обновлением настраиваемых весовых коэффициентов W_z, U_z, W_r, U_r, W и U (помимо опущенного тут члена смещения).

Не волнуйтесь, если вам пока что не все ясно. В конце концов, интуитивное объяснение GRU, приведенное в нескольких предыдущих абзацах, не так уж важно. Понимание во всех подробностях обработки последовательных данных с помощью GRU не требуется обычному инженеру, так же как ему не нужно понимать все нюансы преобразования сверточной сетью входного изображения в выходные вероятности классов. Найденные нейронной сетью в пространстве гипотез подробности в точности отражаются структурой данных RNN в ходе ориентированного на данные процесса обучения.

Для применения GRU к задаче предсказания температуры создадим модель TensorFlow.js, включающую слой GRU (листинг 9.4). Используемый для этого код (вырезан из `jena-weather/train-rnn.js`) практически идентичен взятому нами для модели simpleRNN (см. листинг 9.2). Единственное отличие — тип первого слоя модели (GRU вместо simpleRNN).

Листинг 9.4. Создание GRU-модели для задачи предсказания температуры Jena-weather

```
function buildGRUModel(inputShape) {
  const model = tf.sequential();
  const rnnUnits = 32;
  model.add(tf.layers.gru({
    units: rnnUnits,
    inputShape
  }));
  model.add(tf.layers.dense({units: 1}));
  return model;
}
```

Это зашитое в код количество нейронов, демонстрирующее хорошие результаты, определено путем подбора гиперпараметра вручную

Первый слой модели — слой GRU

Наша модель завершается плотным слоем с одним нейроном и линейной функцией активации по умолчанию для задачи регрессии

Для запуска обучения GRU-модели на наборе данных Jena-weather выполните:

```
yarn train-rnn --modelType gru
```

На рис. 9.5 показаны кривые потерь на обучающем и проверочном наборах данных, полученные при использовании модели на основе GRU. Ее наилучшая погрешность при проверке равна примерно 0,266 — это лучше, чем у модели на основе simpleRNN в предыдущем разделе (0,271). Этот результат отражает бóльшие разрешающие возможности GRU в смысле усвоения последовательных закономерностей по сравнению с simpleRNN. А в показаниях метеорологических инструментов, безусловно, скрыты связанные с упорядоченностью закономерности, позволяющие повысить степень безошибочности предсказания температуры, GRU эту информа-

цию улавливает, а simpleRNN — нет. Но за это приходится платить большей длительностью обучения. Например, на одной из наших машин модель GRU обучается со скоростью 3000 мс/батч, а simpleRNN — 950 мс/батч[1]. Но если цель — предсказать температуру как можно точнее, эти затраты, вероятно, будут оправданны.

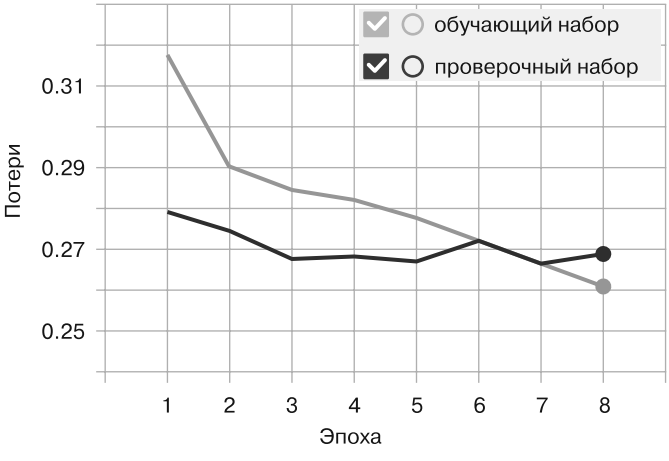


Рис. 9.5. Кривые потерь при обучении модели GRU для задачи предсказания температуры. Сравните с кривыми потерь модели simpleRNN (см. рис. 9.3), и вы заметите, что модель GRU демонстрирует небольшое, но несомненное снижение максимального показателя потерь на проверочном наборе данных

9.2. Создание моделей глубокого обучения для обработки текста

В задаче предсказания температуры мы имели дело с последовательными числовыми данными. Но наиболее распространенный вид последовательных данных, вероятно, все же текст, а не числа. В алфавитных языках, таких как английский, текст можно рассматривать либо как последовательность символов, либо как последовательность слов. Эти два подхода пригодны для различных задач, что мы и покажем в этом разделе. Модели глубокого обучения для текстовых данных, с которыми познакомим вас в следующих разделах, способны решать следующие задачи обработки текста.

- Определение тональности текста (например, того, позитивен или негативен обзор какого-либо товара).
- Классификация текста по тематике (например, выяснение, о чем говорится в новостной статье: политике, финансах, спорте, здоровье, погоде или прочих вопросах).

[1] Эти показатели получены в ходе работы tfjs-node на прикладной части на основе CPU. При использовании tfjs-node-gpu и прикладной части CUDA GPU обе модели будут обучаться пропорционально быстрее.

- Преобразование текстовых входных данных в текстовые выходные данные, (например, для приведения к единому формату или выполнения машинного перевода).
- Предсказание следующих частей текста (например, для интеллектуального ввода текста в смартфонах).

Этот список — лишь малая доля интересных задач машинного обучения, связанных с обработкой текста и изучаемых дисциплиной «обработка естественного языка». И хотя мы затронем лишь краешек методик обработки естественного языка на основе нейронных сетей, идеи и понятия, с которыми вы тут познакомитесь, — неплохая отправная точка для дальнейшего изучения (см. раздел «Материалы для дальнейшего изучения» в конце главы).

Учтите, что ни одна из нейронных сетей, рассматриваемых в этой главе, не способна понимать текст или язык так, как человек, они просто отображают статистическую структуру текста в определенное целевое пространство — непрерывное пространство тональностей, результаты многоклассовой классификации или новую последовательность. Оказывается, что этого достаточно для решения многих задач обработки текста, встречающихся на практике. Глубокое обучение в сфере обработки естественного языка представляет собой просто распознавание паттернов применительно к символам и словам, подобно тому как машинное зрение на основе глубокого обучения (глава 4) — распознавание паттернов применительно к пикселам.

Прежде чем углубиться в предназначенные для обработки текста глубокие нейронные сети, разберемся с представлением текста в машинном обучении.

9.2.1. Представление текста в машинном обучении: унитарное и федеративное кодирование

Большая часть встречавшихся нам в этой книге входных данных — непрерывные. Например, длины лепестков ирисов меняются в определенном непрерывном диапазоне, показания метеорологических приборов в наборе данных Jena-weather — вещественные числа. Эти значения очевидным образом представляются в виде тензоров с плавающей точкой (чисел с плавающей точкой). С текстом все иначе. Текстовые данные поступают в виде строковых значений, состоящих из символов или слов, а не вещественных чисел. Символы и слова дискретны. Например, между j и k не существует никакой буквы, как существует число между 0,13 и 0,14. В этом смысле символы и слова аналогичны классам в многоклассовой классификации (трем видам ирисов или 1000 выходных классов MobileNet). Перед вводом в модели глубокого обучения текстовые данные необходимо преобразовать в векторы (массивы чисел). Такое преобразование называется *векторизацией текста* (text vectorization).

Существует несколько способов векторизации текста. Один из них — *унитарное кодирование* (с ним мы познакомились в главе 3). В английском языке около 10 000 чаще всего используемых слов (в зависимости от того, как считать, конечно). Мы сформируем из них *словарь* (vocabulary), в котором уникальные слова можно отсортировать в определенном порядке (например, в порядке уменьшения частоты

встречаемости), так что в соответствие каждому слову будет поставлен целочисленный индекс[1]. После этого можно будет представить каждое английское слово в виде вектора длиной 10 000, в котором только соответствующий индексу элемент будет равен 1, а все остальные — 0. Это и называется *унитарным представлением* данного слова (рис. 9.6).

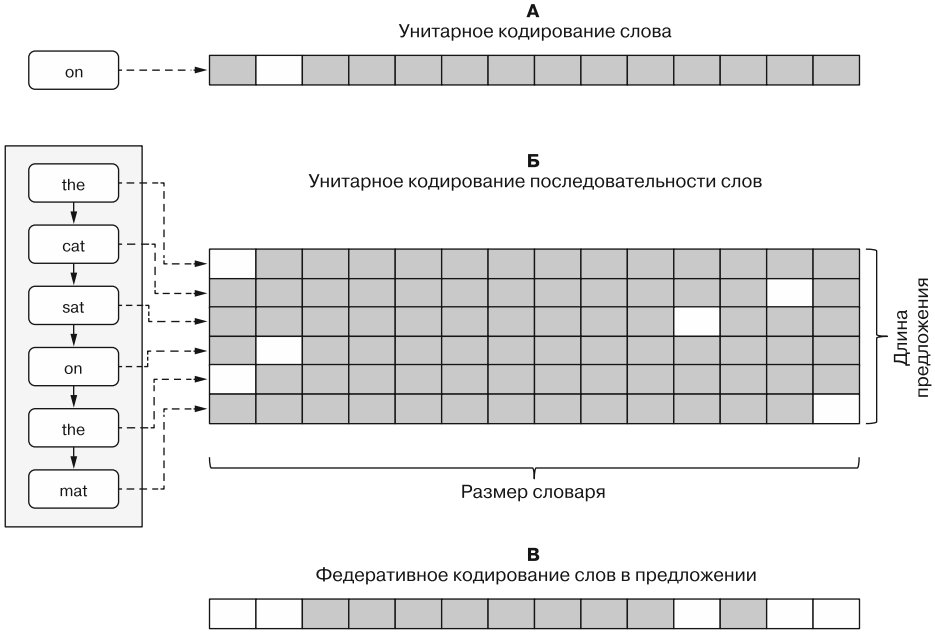


Рис. 9.6. Унитарное кодирование (векторизация) слова (блок А) и предложения как последовательности слов (блок Б). В блоке В приведено упрощенное федеративное представление того же предложения, что и в блоке Б. С его помощью можно представить это предложение более сжатым и лучше масштабируемым образом, но за счет потери информации об упорядоченности. Для удобства визуализации возьмем словарь всего из 14 слов. В практике глубокого обучения используются значительно бóльшие словари (из тысяч или десятков тысяч слов, скажем 10 000)

Но что, если нужно закодировать предложение, а не отдельное слово? Можно вычислить унитарные векторы всех составляющих данное предложение слов и сформировать из них двумерное представление слов предложения (см. блок Б на рис. 9.6). Этот подход прост и однозначен и полностью сохраняет информацию о том, какие

[1] Очевидный вопрос: что, если нам встретится редкое слово, не входящее в словарь из 10 000 слов? С этой проблемой сталкивается каждый алгоритм глубокого обучения, предназначенный для обработки текста. На практике будем решать эту проблему добавлением специального элемента OOV (out-of-dictionary — «не входит в словарь»). Все не входящие в словарь редкие слова группируются в этом специальном элементе и кодируются одним унитарным вектором или вектором вложения. В более сложных методиках используются несколько «корзин» OOV и хеш-функция для распределения по ним редких слов.

слова встречаются в предложении и в каком порядке[1]. Однако для длинного текста размер такого вектора окажется неприемлемо большим. Например, в английском языке предложение в среднем содержит 18 слов. Если словарь состоит из 10 000 слов, представление одного-единственного предложения потребует 180 000 чисел — намного большего объема памяти, чем занимает само предложение. И это не говоря о том, что в некоторых задачах обработки текста приходится иметь дело с абзацами или целыми статьями, состоящими из намного большего числа слов, в результате чего размер представления и объем вычислений вырастет до совершенно неприличных размеров.

Один из способов решения этой проблемы — включить все слова в один вектор, элементы которого отражают наличие/отсутствие соответствующего слова в тексте (см. блок B на рис. 9.6). В этом представлении значение 1 может быть у нескольких элементов вектора. Именно поэтому иногда его называют *федеративным кодированием* (multi-hot encoding). Федеративное кодирование отличается фиксированной длиной, равной размеру словаря, вне зависимости от длины текста, а значит, решает проблему роста размера, правда, за счет потери информации о порядке слов: по федеративному вектору невозможно определить, какие слова шли за какими в тексте. Для некоторых задач это неважно, но для некоторых других такой вариант неприемлем. Существуют более изощренные представления, которые решают проблему роста размера, сохраняя информацию о порядке. Мы обсудим их далее в этой главе. Но сначала взглянем на конкретную задачу машинного обучения, связанную с обработкой текста, которую можно решить с удовлетворительной точностью с помощью федеративного подхода.

9.2.2. Первая попытка анализа тональностей

В первом примере применения машинного обучения к тексту воспользуемся набором данных интернет-базы кинофильмов (IMDb). Он представляет собой набор из примерно 25 000 текстовых обзоров фильмов с сайта imdb.com, маркированных как позитивные или негативные. Задача машинного обучения состоит в бинарной классификации, то есть в определении того, является конкретный обзор фильма позитивным или негативным. Набор данных симметричен (50 % позитивных обзоров и 50 % негативных). Как и следует ожидать от онлайн-обзоров, длины примеров данных сильно различаются. Некоторые из них состоят всего из десяти слов, а другие могут достигать 2000 слов. Вот пример типичного обзора, маркированного как негативный. Пунктуация в этом наборе данных опущена:

the mother in this movie is reckless with her children to the point of neglect i wish i wasn't so angry about her and her actions because i would have otherwise enjoyed the flick what a number she was take my advise and fast forward through everything you see her do until the end also is anyone else getting sick of watching movies that are filmed so dark anymore one can hardly see what is being filmed as an audience we are impossibly involved with the actions on the screen so then why the hell can't we have night vision

[1] В предположении, что в нем нет OOV-слов.

Данные разделены на обучающий и проверочный наборы данных, которые автоматически скачиваются с сайта и записываются в каталог `tmp` при выполнении команды обучения модели:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/sentiment
yarn
yarn train multihot
```

Если внимательно взглянуть на файл `sentiment/data.js`, вы увидите, что скачанные файлы данных не содержат собственно слов в виде строк символов. Слова в них представлены в виде 32-битных целых чисел. И хотя мы не будем подробно описывать код загрузки данных из этого файла, не помешает продемонстрировать в листинге 9.5 код федеративной векторизации предложений.

Листинг 9.5. Федеративная векторизация предложений из функции loadFeatures()

```
const buffer = tf.buffer([sequences.length, numWords]);
sequences.forEach((seq, i) => {
  seq.forEach(wordIndex => {
    if (wordIndex !== OOV_INDEX) {
      buffer.set(1, i, wordIndex);
    }
  });
});
```

Проходим в цикле по всем примерам данных, то есть предложениям

Каждая последовательность (предложение) представляет собой массив целых чисел

Пропускаем OOV-слова для федеративного кодирования

Создает объект TensorBuffer вместо тензора, поскольку далее мы собираемся задавать значения его элементов. Данный буфер изначально заполнен нулями

Устанавливаем в 1 соответствующий индекс в буфере. Учтите, что каждому индексу i может соответствовать несколько установленных в 1 значений wordIndex, отсюда и название «федеративное кодирование»

Федеративно кодированные признаки представлены в виде двумерного тензора формы `[numExamples, numWords]`, где `numWords` — размер словаря (в нашем случае 10 000). Эта форма не зависит от длины отдельных предложений, что сильно упрощает парадигму векторизации. Форма тензора загружаемых из файлов целевых признаков — `[numExamples, 1]`, он включает негативные и позитивные метки в виде нулей и единиц соответственно.

К этим федеративным данным применим модель MLP. На самом деле мы никак не смогли бы применить к этим данным RNN, даже если бы хотели, поскольку при федеративном кодировании информация о порядке была утрачена. Мы поговорим о подходах на основе RNN в следующем разделе. Код создания MLP-модели, взятый из функции `buildModel()` в файле `sentiment/train.js`, выглядит (с некоторыми упрощениями) так, как показано в листинге 9.6.

Выполнив команду `yarn train multihot --maxLen 500`, вы увидите, что модель достигает наилучшей степени безошибочности на проверочном наборе примерно 0,89. Это вполне приемлемая степень безошибочности, она существенно выше, чем при случайном гадании (0,5). А значит, для достижения приемлемой степени безошибочности вполне достаточно учитывать лишь то, какие слова встречаются в обзоре. Например, слова наподобие *enjoyable* (доставляющий удовольствие) и *sublime* (потрясающий) ассоциируются с позитивными обзорами, а слова наподобие *sucks*

(отстой) и *bland* (примитив) — с негативными с довольно высокой степенью достоверности. Конечно, существует множество случаев, когда учет только того, из каких слов состоит обзор, приводит к ложным выводам. В качестве примера рассмотрим предложение *Don't get me wrong, I hardly disagree this is an excellent film*, для выяснения истинного смысла которого необходимо учесть информацию о последовательности — не только сами слова, но и порядок, в котором они стоят в предложении. В следующем разделе мы покажем, что с помощью векторизации текста, при которой информация о последовательности не отбрасывается, и модели, способной эту информацию учитывать, можно превзойти указанную эталонную степень безошибочности. Давайте взглянем теперь, что представляют собой вложения слов и как работают одномерные сверточные сети.

Листинг 9.6. Создание MLP-модели для федеративно кодированных обзоров фильмов IMDb

```
const model = tf.sequential();
model.add(tf.layers.dense({
  units: 16,
  activation: 'relu',
  inputShape: [vocabularySize]
}));
model.add(tf.layers.dense({
  units: 16,
  activation: 'relu'
}));
model.add(tf.layers.dense({
  units: 1,
  activation: 'sigmoid'
}));
```

Добавляем два скрытых слоя с функцией активации ReLU для повышения разрешающих возможностей модели

Форма входного сигнала соответствует размеру словаря благодаря федеративной векторизации

Для выходного слоя используется сигма-функция активации в соответствии с требуемым для задачи бинарной классификации

9.2.3. Более эффективное представление текста — вложения слов

Что такое *вложение слов* (word embedding)? Подобно унитарному кодированию (см. рис. 9.6), вложение слов — способ представления слов в виде векторов (одномерных тензоров в TensorFlow.js). Однако при использовании вложений слов можно усваивать значения элементов векторов, а не жестко «зашивать» их в соответствии со строгими правилами наподобие карты соответствий слов индексам в унитарном представлении. Другими словами, когда применяется ориентированная на обработку текста нейронная сеть вложений слов, векторы этих вложений становятся обучаемыми весовыми параметрами модели и обновляются посредством того же процесса обратного распространения, что и прочие весовые параметры модели.

Эта ситуация схематически изображена на рис. 9.7. Для работы с вложениями слов в TensorFlow.js предназначен тип слоя `tf.layer.embedding()`, содержащий обучаемую матрицу весов формы `[vocabularySize, embeddingDims]`, где `vocabularySize` — число уникальных слов в словаре, а `embeddingDims` — выбранная пользователем размерность векторов вложений. При получении слова, допустим, *the* мы находим с помощью поисковой таблицы соответствий слов индексам нужную

строку в матрице вложений, которая и будет вектором вложений для данного слова. Учтите, что поисковая таблица соответствий слов индексам не является частью слоя вложений — ее необходимо поддерживать в виде отдельной от модели сущности (см. пример в листинге 9.9).

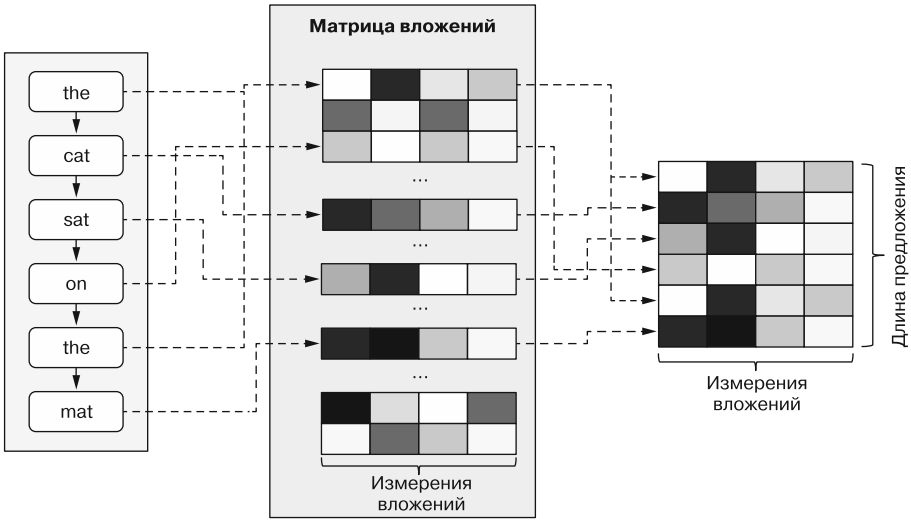


Рис. 9.7. Схематическая иллюстрация функционирования матрицы вложений. Каждой строке матрицы вложений соответствует слово в словаре, а столбцу — одно из измерений вложений. Значения элементов матрицы вложений, отраженные на схеме различными оттенками серого, выбраны случайным образом

Для последовательности слов, например, показанного на рис. 9.7 предложения необходимо повторить процесс поиска для всех слов в правильном последовательном порядке и сформировать из полученных в результате векторов вложений двумерный тензор формы `[sequenceLength, embeddingDims]`, где `sequenceLength` — число слов в предложении[1]. А что, если предложение включает повторяющиеся слова, не имеющие никакого значения, наподобие слова *the* в примере на рис. 9.7? Просто несколько раз включите в итоговый двумерный тензор один и тот же вектор вложений.

Вот несколько преимуществ вложений слов.

- Они позволяют решить проблему размера унитарных представлений. `embeddingDims` обычно намного меньше, чем `vocabularySize`. Например, в одномерной сверточной сети, которую мы собираемся применить к набору данных IMDb, `vocabularySize` составляет 10 000, а `embeddingDims` — всего 128. Так что представление содержащего 500 слов обзора фильма из набора данных IMDb потребует всего $500 \times 128 = 64$ тысяч чисел с плавающей точкой, вместо $500 \times 10\,000 = 5\,000\,000$ чисел в случае унитарных представлений — намного более экономичная векторизация.

[1] Для эффективного поиска вложений многих слов можно воспользоваться методом tf.gather(), с помощью которого скрыто реализован слой вложений в TensorFlow.js.

- Вложения слов способны усваивать семантические связи между словами, поскольку не требуют жесткого упорядочения слов в словаре и дают возможность усвоения матрицы вложений с помощью обратного распространения ошибки, подобно всем прочим весовым коэффициентам нейронной сети. Векторы вложений слов с близкими значениями должны быть близки в пространстве вложений. Например, векторы слов со схожим смыслом *very* и *truly* должны быть ближе, чем у более далеких по смыслу слов *very* и *barely*. Почему? Интуитивно это можно понять так: если заменить несколько слов в обзоре фильма словами со схожим смыслом, хорошо обученная нейронная сеть должна выдать тот же результат классификации. Но это возможно лишь тогда, когда векторы вложений для всех соответствующих пар слов, представляющих собой входные данные для соответствующей части модели, близки друг к другу.
- Кроме того, благодаря многомерности (например, 128-мерности) пространства вложений векторы вложений могут захватывать различные аспекты слов. Например, одно измерение может отражать часть речи, и в нем прилагательное *fast* ближе к прочим прилагательным (таким как *warm*), чем к существительным (например, *house*). А другое измерение может кодировать род слова, и в нем слово *actress* будет ближе к другому слову женского рода (такому как *queen*), чем к любому слову мужского рода (например, *actor*). В следующем разделе (см. инфобокс 9.2) мы покажем, как визуализировать вложения слов и исследовать интересные структуры, возникающие после обучения основанной на вложениях слов нейронной сети на наборе данных IMDb.

В табл. 9.1 приведена краткая сводка различий между унитарным/федеративным представлением и вложениями слов — двумя чаще всего используемыми парадигмами векторизации слов.

Таблица 9.1. Сравнение двух парадигм векторизации слов — унитарного/федеративного кодирования и вложения слов

	Унитарное/федеративное представление	**Вложения слов**
Усваивается моделью или «зашивается» в код?	«Зашивается» в код	Усваиваются: матрица вложений представляет собой обучаемый весовой параметр, ее значения часто отражают семантическую структуру словарей после обучения
Разреженное или плотное?	Разреженное: большинство элементов нулевые, некоторые равны 1	Плотное: элементы принимают непрерывные значения
Масштабируемость	Не масштабируется на словари большого размера, размер вектора пропорционален размеру словаря	Масштабируется на словари большого размера, размер вложений (число измерений вложений) не обязательно растет с увеличением количества слов в словаре

9.2.4. Одномерные сверточные сети

В главе 4 мы показали, что двумерные сверточные слои играют ключевую роль в глубоких нейронных сетях, входными данными в которых являются изображения. Слои conv2d можно обучить представлять локальные признаки на маленьких двумерных фрагментах изображений. Понятие свертки можно распространить на последовательности. В результате получается алгоритм *одномерной свертки* (1D convolution), доступный в TensorFlow.js в виде функции `tf.layers.conv1d()`. В основе слоев conv2d и conv1d лежит одна идея: они оба представляют собой обучаемые средства выделения трансляционно инвариантных локальных признаков. Например, после обучения на изображениях слой conv2d может стать чувствительным к паттернам углов определенной ориентации и определенного цвета, а слой conv2d может стать чувствительным к паттерну «негативный глагол, за которым следует одобрительное прилагательное» после обучения на тексте[1].

На рис. 9.8 подробно показана схема работы слоя conv1d. Как вы помните из рис. 4.3, в слое conv2d ядро скользит по всем возможным позициям входного изображения. Алгоритм одномерной свертки также включает в себя скольжение ядра, но более простое, в одном измерении. При таком скольжении на каждой позиции извлекается срез входного тензора. Длина среза равна `kernelSize` (задаваемое поле конфигурации слоя conv1d), и в нашем случае у него есть второе

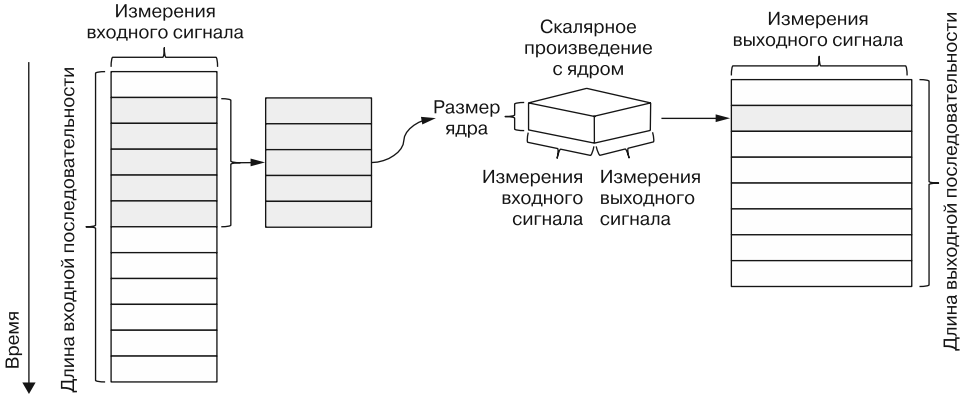


Рис. 9.8. Схематическая иллюстрация работы одномерной свертки (tf.layers.conv1d()). Для простоты показан только один входной пример *(слева)*. Мы предполагаем, что длина входной последовательности равна 14, а размер ядра слоя conv1d — 5. Для каждой из позиций скользящего окна выделяется срез входной последовательности длиной 5. Этот срез скалярно умножается на ядро слоя conv1d, и получается отдельный срез выходной последовательности. В результате повторения этого процесса для всех возможных позиций скользящего окна получается выходная последовательность *(справа)*

[1] Как вы, наверное, догадались, существует и трехмерная свертка, удобная для задач глубокого обучения, связанных с трехмерными (пространственными) данными, например определенными типами медицинских снимков и геологических данных.

измерение, равное числу измерений вложений. Далее вычисляется *внутреннее произведение* (умножение и сложение) входного среза и ядра слоя conv1d, дающее в результате один срез выходной последовательности. Эта операция повторяется для всех возможных позиций скольжения до тех пор, пока не будет сгенерирован полный выходной сигнал. Как и входной тензор слоя conv1d, полный выходной сигнал представляет собой последовательность, хотя и другой длины (определяемой длиной входной последовательности, `kernelSize` и другими настройками слоя conv1d) и с другим числом измерений признаков (определяемых параметром `filters` конфигурации слоя conv1d). Благодаря этому можно сформировать из нескольких идущих подряд слоев conv1d глубокую одномерную сверточную сеть аналогично тому, как в двумерных сверточных сетях часто размещается подряд несколько слоев conv2d.

Усечение последовательностей и дополнение их символами

Теперь, когда мы уже умеем применять слои conv1d для машинного обучения на основе текстов, готовы ли мы обучить одномерную сверточную сеть на данных IMDb? Не совсем. Осталось рассказать про усечение и дополнение последовательностей. Зачем нужны операции усечения и дополнения? Для моделей TensorFlow.js требуются входные данные `fit()` в виде тензоров, а тензор должен иметь четкую форму. Следовательно, хотя длины обзоров фильмов не фиксированы (напомним, что они могут варьироваться от 10 до 2400 слов), мы должны выбрать конкретное значение длины в качестве второго измерения входного тензора признаков (`maxLen`), чтобы получить полную форму входного тензора вида `[numExamples, maxLen]`. При использовании федеративного кодирования в предыдущем разделе такой проблемы не было, поскольку длина последовательности не влияет на второе измерение тензоров федеративного представления.

Выбор значения `maxLen` определяется следующими соображениями.

- Она должна быть достаточно велика для захвата полезной части большинства обзоров. Например, значение `maxLen = 20`, вероятно, окажется слишком маленьким и наиболее ценные части большинства обзоров будут отброшены.
- Она не должна быть намного больше длины большинства обзоров, чтобы не расходовать память и вычислительные ресурсы впустую.

В качестве компромисса мы выбрали для этого примера значение 500 (максимум) слов на обзор. Оно задается с помощью флага `--maxLen` команды обучения одномерной сверточной сети:

```
yarn train --maxLen 500 cnn
```

После выбора `maxLen` необходимо привести к ней все примеры обзоров: более длинные обрезать, а более короткие дополнить. Именно для этого служит функция `padSequences()` из листинга 9.7. Существует два способа усечения слишком длинной

последовательности: отбрасывание ее начала (опция `'pre'`) или концовки. В данном случае применяем первый вариант из тех соображений, что концовка обзора фильма скорее содержит относящуюся к тональности информацию, чем начало. Аналогично увеличить слишком короткую последовательность до нужной длины также можно двумя способами: добавить нужное количество заполняющих символов (`PAD_CHAR`) до (опция `'pre'`) или после последовательности. Здесь мы волевым решением также выбрали первый вариант. Код листинга взят из файла `sentiment/sequence_utils.js`.

Листинг 9.7. Усечение и дополнение последовательности как один из шагов загрузки текстовых признаков

```
export function padSequences(
    sequences, maxLen,
    padding = 'pre',
    truncating = 'pre',
     value = PAD_CHAR) {
  return sequences.map(seq => {
    if (seq.length > maxLen) {
      if (truncating === 'pre') {
        seq.splice(0, seq.length - maxLen);
      } else {
        seq.splice(maxLen, seq.length - maxLen);
      }
    }

    if (seq.length < maxLen) {
      const pad = [];
      for (let i = 0; i < maxLen - seq.length; ++i) {
        pad.push(value);
      }
      if (padding === 'pre') {
        seq = pad.concat(seq);
      } else {
        seq = seq.concat(pad);
      }
    }

    return seq;
  });
}
```

Проходим в цикле по всем входным последовательностям

Данная конкретная последовательность превышает предусмотренную длину (maxLen), так что выполняем ее усечение до maxLen

Существует два способа усечения последовательности: отбросить ее начало ('pre') или концовку

Данная последовательность короче предусмотренной длины: ее необходимо дополнить

Генерируем дополняющую последовательность

Аналогично усечению, дополнить слишком короткую последовательность также можно двумя способами: спереди ('pre') или сзади

Примечание: seq, длина которой в точности равна maxLen, возвращается без изменений

Создание и запуск одномерной сверточной сети для набора данных IMDb

Все составные части нашей одномерной сверточной сети готовы, давайте соберем их вместе и посмотрим, удастся ли добиться большей безошибочности при анализе тональностей IMDb. Код создания одномерной сверточной сети приведен в листинге 9.8 (фрагмент из файла `sentiment/train.js` с некоторыми упрощениями). Затем дана сводка топологии получившегося объекта `tf.Model`.

Листинг 9.8. Создание одномерной сверточной сети для задачи IMDb

```
const model = tf.sequential();
model.add(tf.layers.embedding({
  inputDim: vocabularySize,
  outputDim: embeddingSize,
  inputLength: maxLen
}));
model.add(tf.layers.dropout({rate: 0.5}));
model.add(tf.layers.conv1d({
  filters: 250,
  kernelSize: 5,
  strides: 1,
  padding: 'valid',
  activation: 'relu'
}));
model.add(tf.layers.globalMaxPool1d({}));
model.add(tf.layers.dense({
  units: 250,
  activation: 'relu'
}));
model.add(tf.layers.dense({units: 1, activation: 'sigmoid'}));
```

Наша модель начинается со слоя вложений, преобразующего входные целочисленные индексы в соответствующие векторы слов

Слою вложений необходима информация о размере словаря. Без этого он не сможет определить размер матрицы вложений

Добавляем слой дропаута для борьбы с переобучением

А вот и слой conv1d

Слой globalMaxPool1d схлопывает измерение времени путем выделения в каждом из фильтров максимального элемента. Его выходной сигнал подходит для использования в качестве входного сигнала последующих плотных слоев (MLP)

Добавляем поверх модели состоящий из двух слоев MLP

```
_________________________________________________________________
Layer (type)                 Output shape              Param #
=================================================================
embedding_Embedding1 (Embedd [null,500,128]            1280000
_________________________________________________________________
dropout_Dropout1 (Dropout)   [null,500,128]            0
_________________________________________________________________
conv1d_Conv1D1 (Conv1D)      [null,496,250]            160250
_________________________________________________________________
global_max_pooling1d_GlobalM [null,250]                0
_________________________________________________________________
dense_Dense1 (Dense)         [null,250]                62750
_________________________________________________________________
dense_Dense2 (Dense)         [null,1]                  251
=================================================================
Total params: 1503251
Trainable params: 1503251
Non-trainable params: 0
_________________________________________________________________
```

Полезно будет одновременно взглянуть на JavaScript-код и текстовую сводку топологии модели. Здесь стоит отметить несколько нюансов.

- Форма модели `[null, 500]`, где `null` — неопределенное измерение батчей (число примеров данных), а 500 — максимально допустимая длина обзора в словах (`maxLen`). Входной тензор содержит усеченные/дополненные последовательности целочисленных индексов слов.
- Первый слой модели — слой вложений, преобразующий индексы слов в соответствующие векторы слов, в результате чего получается форма `[null, 500, 128]`.

Как видите, длина последовательности (500) сохраняется, а последний элемент формы отражает измерение вложений (128).

- За слоем вложений следует слой conv1d — главная часть модели. Размер его ядра выбран равным 5, шаг свертки по умолчанию — 1 и дополнение — `'valid'`. В результате есть 500 – 5 + 1 = 496 возможных позиций скользящего окна измерения последовательностей. Так что второй элемент формы выходного сигнала (`[null, 496, 250]`) равен 496. Последний элемент его формы отражает количество фильтров в слое conv1d.
- Слой globalMaxPool1d, следующий за слоем conv1d, чем-то напоминает слой maxPooling2d, встречавшийся в сверточных сетях для изображений. Впрочем, он производит более радикальную субдискретизацию, при которой все элементы измерения последовательностей схлопываются в одно максимальное значение. В результате получаем форму выходного сигнала `[null, 250]`.
- Добившись одномерной формы тензора (не считая измерения батчей), мы можем добавить поверх него два плотных слоя и таким образом сформировать MLP поверх всей модели.

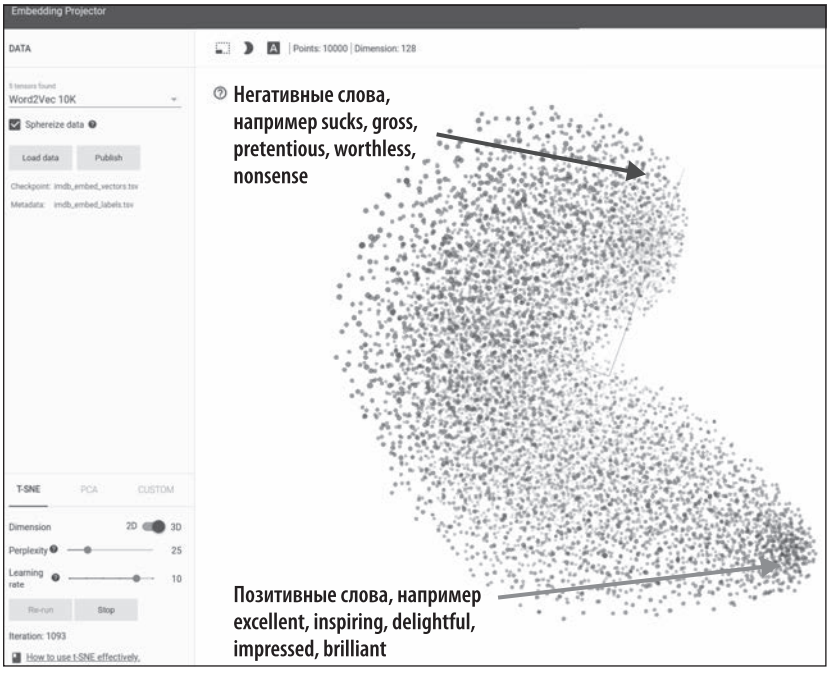
Запустите обучение одномерной сверточной сети с помощью команды `yarn train --maxLen 500 cnn`. После двух-трех эпох обучения вы увидите, что модель достигла максимальной безошибочности около 0,903 на проверочном наборе данных, — это небольшой, но весомый прирост по сравнению со степенью безошибочности MLP, основанного на федеративной векторизации (0,890). Он отражает усвоенную одномерной сверточной сетью информацию о последовательной упорядоченности, которую никак не мог усвоить федеративный MLP.

Как же одномерная сверточная сеть захватывает информацию о последовательной упорядоченности? С помощью сверточного ядра. Скалярное произведение этого ядра чувствительно к упорядоченности элементов. Например, если входной пример данных состоит из пяти слов, *I like it so much*, одномерная свертка выдаст одно конкретное значение, однако, если поменять порядок слов на *much so I like it*, результат одномерной свертки изменится, хотя набор элементов останется тем же самым.

Однако следует отметить, что слой conv1d сам по себе не способен усваивать последовательные паттерны, выходящие за размеры его ядра. Например, представьте себе, что на смысл предложения влияет порядок двух расположенных далеко друг от друга слов: слой conv1d с ядром, размер которого меньше расстояния между ними, не сможет усвоить такое «далекое» взаимодействие. В этом отношении RNN, такие как GRU и LSTM, превосходят одномерную свертку.

Один из способов устранения этого недостатка для одномерных сверточных сетей — углубление, а именно наращивание числа последовательных слоев conv1d, чтобы «рецептивное поле» высокоуровневых слоев conv1d было достаточно большим для захвата подобных «далеких» зависимостей. Впрочем, во многих задачах машинного обучения, связанных с обработкой текста, «далекие» зависимости не играют важной роли, так что достаточно одномерной сверточной сети с небольшим числом слоев conv1d. Можете попробовать в примере анализа тональностей IMDb обучить

ИНФОБОКС 9.2. Использование Embedding Projector для визуализации усвоенных векторов вложений



Визуализация усвоенных вложений слов из одномерной сверточной сети с помощью уменьшения размерности посредством t-SNE в Embedding Projector

Появились ли какие-то интересные структуры во вложениях слов одномерной сверточной сети после обучения? Чтобы узнать это, можете воспользоваться необязательным флагом `--embeddingFilesPrefix` команды `yarn train`:

```
yarn train --maxLen 500 cnn --epochs 2 --embeddingFilesPrefix /tmp/imdb_embed
```

Эта команда генерирует два файла:

- `/tmp/imdb_embed_vectors.tsv` — файл разделенных символами табуляции числовых значений вложений слов. Каждая строка содержит вектор вложений для какого-либо слова. В нашем случае в файле 10 000 строк (в соответствии с размером словаря), а в каждой строке — 128 чисел (измерения вложений);
- `/tmp/imdb_embed_labels.tsv` — файл, содержащий метки слов, соответствующие векторам предыдущего файла. Каждая строка состоит из одного слова.

Эти файлы можно загрузить в Embedding Projector (https://projector.tensorflow.org/) для визуализации (см. предыдущий рисунок). А поскольку наши векторы вложений относятся к многомерному (а именно, 128-мерному) пространству, необходимо понизить их размерность до трех или менее измерений, чтобы они были понятны людям.

Утилита Embedding Projector предоставляет два алгоритма понижения размерности: метод стохастических вложений соседей на основе распределения Стьюдента (t-SNE) и метод главных компонент (PCA), которые мы не станем обсуждать во всех подробностях. Но если вкратце, эти методы отображают многомерные векторы вложений в трехмерное пространство с минимизацией потерь в связях между этими векторами. t-SNE — более сложный и требующий бóльших вычислительных затрат метод из этих двух. Полученная в результате его применения визуализация показана на рисунке.

Каждая точка в облаке точек соответствует одному из слов словаря. Подвигайте указателем мыши и задерживайте его над точками, чтобы увидеть, каким словам они соответствуют. Наши векторы вложений, обученные на этом небольшом наборе данных для анализа тональностей, уже демонстрируют определенные интересные структуры, отражающие семантику слов. В частности, на одном из концов облака точек довольно много слов, часто встречающихся в позитивных обзорах фильмов (например, *excellent*, *inspiring* и *delightful*), а на противоположном конце содержится множество негативно звучащих слов (*sucks*, *gross* и *pretentious*). При обучении бóльших моделей на больших массивах текста могут обнаружиться и другие интересные структуры, но и из этого маленького примера понятно, насколько широки возможности метода вложений слов.

Поскольку вложения слов — важная часть ориентированных на обработку текста глубоких нейронных сетей, исследователи создали готовые предварительно обученные вложения слов, благодаря которым специалистам-практикам больше не приходится обучать собственные вложения слов, как мы делали в примере сверточной сети IMDb. Один из наиболее известных наборов предварительно обученных вложений слов — GloVe (от Global Vectors — глобальные векторы), созданный группой по обработке естественного языка Стэнфордского университета (см. https://nlp.stanford.edu/projects/glove/).

У использования предварительно обученных вложений слов наподобие GloVe — двойное преимущество. Во-первых, они уменьшают объем необходимых во время обучения вычислений, поскольку слой вложений не нужно обучать далее, а значит, его можно просто заблокировать. Во-вторых, предварительно обученные вложения наподобие GloVe усваиваются на миллиардах слов, а поэтому намного лучше всего, чего можно добиться при обучении на маленьком наборе данных, таком как наш набор данных IMDb. В этом смысле роль предварительно обученных вложений слов в задачах обработки естественного языка аналогична роли баз предварительно обученных глубоких сверточных сетей (таких как MobileNet, которую мы видели в главе 5) в машинном зрении.

модель на основе LSTM с теми же значением `maxLen` и измерениями вложений, с которыми обучали одномерную сверточную сеть:

```
yarn train --maxLen 500 lstm
```

Заметьте, что максимальная безошибочность на проверочном наборе данных LSTM-сети (аналогичной GRU, но чуть более сложной, см. рис. 9.4) примерно такая же, как и у одномерной сверточной сети, вероятно, вследствие того, что «далекие» связи между словами и фразами имеют не слишком большое значение в этом массиве обзоров фильмов и задаче классификации тональностей.

Таким образом, одномерные сверточные сети — заманчивая альтернатива RNN для подобных задач обработки текста, особенно если учесть намного меньшие вычислительные затраты одномерных сверточных сетей по сравнению с RNN. При выполнении командам `cnn` и `lstm` видно, что обучение одномерной сверточной сети происходит почти в шесть раз быстрее обучения LSTM-модели. LSTM и RNN работают медленнее из-за пошагового выполнения внутренних операций, не допускающих распараллеливания. сверточные же сети изначально приспособлены для распараллеливания.

Применения одномерной сверточной сети для выполнения вывода на веб-странице

В файле `sentiment/index.js` вы найдете код, который развертывает обученную в Node.js модель для использования на стороне клиента. Чтобы увидеть клиентское приложение в действии, выполните команду `yarn watch` так же, как в прочих примерах в этой книге. Эта команда скомпилирует код, запустит веб-сервер и автоматически откроет в браузере вкладку со страницей `index.html`. На этой странице вы сможете нажать кнопку для загрузки обученной модели посредством HTTP-запросов и выполнения с ее помощью анализа тональностей обзоров фильмов в окне ввода текста. В этом окне пример обзора фильма можно редактировать, так что вы можете вносить в него любые изменения и наблюдать за их влиянием на бинарные предсказания в режиме реального времени. Страница включает два готовых примера обзоров (позитивный и негативный), которыми вы можете воспользоваться как отправной точкой для своих экспериментов. Загруженная одномерная сверточная сеть работает достаточно быстро для генерации оценок тональностей на лету по мере ввода текста в окне.

Основная часть кода выполнения вывода очевидна (см. листинг 9.9, взятый из файла `sentiment/index.js`), но стоит отметить несколько интересных нюансов.

- До преобразования в индексы слов весь входной текст переводится в нижний регистр, все знаки препинания отбрасываются, а также удаляются лишние пробелы. Это необходимо, поскольку используемый нами словарь содержит слова в нижнем регистре.
- Особый индекс слова (`OOV_INDEX`) ставится в соответствие словам, не входящим в словарь (OOV-словам). В их число входят редкие слова и слова с опечатками.
- Чтобы обеспечить нужную длину входных тензоров модели, используется та же функция `padSequences()`, что и для обучения (см. листинг 9.7). Как мы уже видели, реализуется это посредством усечения и дополнения. Вот наглядный пример преимуществ TensorFlow.js для подобных задач машинного обучения: один и тот же код предварительной обработки можно применять и для среды обучения в прикладной части, и для среды выдачи в клиентской части, снижая тем самым риск возникновения асимметрии данных (листинг 9.9) (более подробно риски асимметрии обсуждаются в главе 6).

Листинг 9.9. Использование обученной одномерной сверточной сети для выполнения вывода в клиентской части

```
predict(text) {
  const inputText =
      text.trim().toLowerCase().replace(/(\.|\,|\!)/g, '').split(' ');
  const sequence = inputText.map(word => {
    let wordIndex =
        this.wordIndex[word] + this.indexFrom;
    if (wordIndex > this.vocabularySize) {
      wordIndex = OOV_INDEX;
    }
    return wordIndex;
  });
  const paddedSequence =
      padSequences([sequence], this.maxLen);
  const input = tf.tensor2d(
      paddedSequence, [1, this.maxLen]);

  const beginMs = performance.now();
  const predictOut = this.model.predict(input);
  const score = predictOut.dataSync()[0];
  predictOut.dispose();
  const endMs = performance.now();

  return {score: score, elapsed: (endMs - beginMs)};
}
```

Преобразуем в нижний регистр; удаляем знаки препинания и лишние пробелы из входного текста

Ставим в соответствие всем словам индексы слов. this.wordIndex загружен из JSON-файла

Не входящие в словарь слова представляются с помощью специального индекса слова: OOV_INDEX

Усечение длинных обзоров и дополнение коротких до нужной длины

Преобразование данных в тензорное представление для подачи на вход модели

Отслеживаем время, потраченное на выполнение вывода на основе нашей модели

Здесь происходит сам вывод (прямой проход модели)

9.3. Решение задач преобразования последовательностей в последовательности с помощью механизма внимания

В примерах прогноза погоды Jena-weather и определения тональностей IMDb мы показали, как можно предсказывать одно число или класс по входной последовательности. Впрочем, часть наиболее интересных задач обработки последовательных данных требуют генерации *выходной последовательности* на основе входной. Подобные задачи называются *задачами преобразования последовательностей в последовательности* (sequence-to-sequence, seq2seq). Существует множество разновидностей задач преобразования последовательностей в последовательности, вот лишь малая их доля.

- *Автоматическое реферирование текста* (text summarization) — генерация сжатого изложения (например, 100 слов или менее) заданного текста, который может содержать десятки тысяч слов.
- *Машинный перевод* — по абзацу на одном языке (например, английском) сгенерировать перевод на другой язык (допустим, японский).

- *Предсказание слов для автодополнения* — предсказать, какие слова могут следовать за заданными несколькими первыми словами предложения. Полезно для автодополнения/рекомендаций в приложениях для работы с электронной почтой и UI для поисковых систем.
- *Сочинение музыки* — по заданной последовательности музыкальных нот сгенерировать начинающуюся с них мелодию.
- *Чат-боты* — генерировать ответы на вводимые пользователем предложения, реализующие определенные цели разговора (например, для поддержки пользователей или просто поддержания разговора).

Механизм внимания (attention mechanism)[1] — мощный и популярный метод решения задач seq2seq, часто применяемый в сочетании с RNN. В этом разделе покажем, как задействовать механизм внимания и LSTM для решения простой задачи seq2seq — преобразования набора дат в календарном формате в стандартный формат дат. И хотя мы специально выбрали очень простой пример, полученные при его рассмотрении знания вполне применимы к более сложным задачам seq2seq наподобие перечисленных ранее. Сначала сформулируем задачу преобразования формата дат.

9.3.1. Постановка задачи преобразования последовательности в последовательность

Наверное, вы, как и мы, часто путаетесь (или даже немного злитесь) из-за множества способов записи календарных дат, особенно когда путешествуете по разным странам. В некоторых предпочитают порядок «месяц, день, год», некоторые — «день, месяц, год», а третьи — «год, месяц, день». И даже при одинаковом порядке существуют варианты: записывать ли месяц с помощью слова (январь), сокращения (янв), числа (1) или дополненного нулем числа из двух цифр (01). День месяца также можно дополнять спереди нулем, записывать в виде порядкового (4-е) или количественного (4) числительного. Что касается года, можно записывать все четыре цифры номера года или только две последние. Более того, при конкатенации к частям, обозначающим год, месяц и день, можно добавлять пробелы, запятые, точки и косые черты или не добавлять никаких промежуточных символов! Всевозможные сочетания этих вариантов дают суммарно как минимум несколько десятков способов записи одной и той же даты.

Так что отнюдь нелишним будет алгоритм, получающий на входе строку календарной даты в одном из этих форматов и возвращающий соответствующую строку даты в формате ISO-8601 (например, 2019-02-05). Эту задачу можно решить и без машинного обучения, путем написания традиционной программы. Но при большом числе возможных форматов это непростая задача, требующая немало времени, и итого-

[1] См.: *Graves A.* Generating Sequences with Recurrent Neural Networks // submitted 4 Aug. 2013, https://arxiv.org/abs/1308.0850; *Bahdanau D., Cho K., Bengio Y.* Neural Machine Translation by Jointly Learning to Align and Translate // submitted 1 Sept. 2014. https://arxiv.org/abs/1409.0473.

вый код может разрастись до нескольких сотен строк. Давайте попробуем реализовать подход на основе глубокого обучения — в частности, архитектуру «кодировщик — декодировщик» с использованием механизма внимания на основе LSTM.

Чтобы ограничить поле охвата этого примера, начнем с 18 часто встречающихся форматов дат, приведенных далее. Отметим, что все они представляют собой разные способы записи одной и той же даты:

```
"23Jan2015", "012315", "01/23/15", "1/23/15",
"01/23/2015", "1/23/2015", "23-01-2015", "23-1-2015",
"JAN 23, 15", "Jan 23, 2015", "23.01.2015", "23.1.2015",
"2015.01.23", "2015.1.23", "20150123", "2015/01/23",
"2015-01-23", "2015-1-23"
```

Конечно, существуют и другие форматы дат[1]. Но когда фундамент в виде обучения модели и выполнения вывода заложен, добавление поддержки дополнительных форматов — простая задача, которую мы оставим читателю в качестве упражнения в конце данной главы (упражнение 3).

Прежде всего запустим пример. Как и пример анализа тональностей, приведенный ранее, он состоит из двух частей: обучения и выполнения вывода. Обучение производится в прикладной части с помощью tfjs-node или tfjs-node-gpu. Для запуска обучения выполните следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/sentiment
yarn
yarn train
```

Чтобы выполнить обучение с помощью GPU с поддержкой CUDA, воспользуйтесь флагом `--gpu` команды `yarn train`:

```
yarn train --gpu
```

По умолчанию обучение продолжается две эпохи, чего должно хватить для приближения значения потерь к нулю, а степени безошибочности преобразования — к идеальной. Большинство результатов вывода для примеров данных, отображаемые в конце обучения, должны оказаться правильно преобразованными. Эти примеры данных для вывода взяты из контрольного набора данных, не пересекающегося с обучающим набором. Обученная модель сохраняется по относительному пути dist/model и затем используется на этапе выполнения вывода в браузере. Для запуска UI вывода выполните:

```
yarn watch
```

На появившейся веб-странице вы можете ввести дату в поле ввода текста Input Date String (Введите строку с датой) и нажать клавишу Enter — и увидите, как изме-

[1] Кроме того, вы могли обратить внимание на то, что мы выбрали набор форматов дат без неоднозначностей. Если бы мы включили в набор форматов варианты и MM/DD/YYYY, и DD/MM/YYYY, некоторые строковые представления дат допускали бы двоякое толкование, то есть их нельзя было бы интерпретировать с уверенностью. Например, строковое значение “01/02/2019” можно интерпретировать и как 2 января 2019 года, и как 1 февраля 2019 года.

нится соответствующим образом выходное строковое представление даты. Кроме того, там в виде карты интенсивности с областями различных оттенков отображается используемая в ходе преобразования матрица внимания (рис. 9.9), что делает интерпретацию особенно удобной для людей. Поэкспериментируйте с ней, чтобы познакомиться лучше.

Рассмотрим в качестве примера показанный на рис. 9.9 результат. Выходной сигнал модели (`"2034-07-18"`) представляет собой правильно преобразованную входную дату (`"JUL 18, 2034"`). Строки матрицы внимания соответствуют входным символам (`"J"`, `"U"`, `"L"`, `" "` и т. д.), а столбцы — выходным символам (`"2"`, `"0"`, `"3"` и т. д.). Таким образом, каждый из элементов матрицы внимания указывает, сколько внимания уделяется входному символу при генерации соответствующего выходного символа. Чем больше значение элемента, тем больше внимания уделяется. Например, взгляните на четвертый столбец последней строки, соответствующий последнему входному символу (`"4"`) и четвертому выходному символу (`"4"`). Как видно по оттенку цвета, его значение довольно велико. Это логично, ведь последняя цифра части выходного сигнала, обозначающей год, и должна в основном зависеть от последней цифры части входного строкового значения, соответствующей году. И напротив, значения остальных элементов этого столбца, расположенные ниже, а значит, генерация символа `"4"` выходного строкового значения практически не используют информацию из остальных символов входной строки. Аналогичные закономерности можно наблюдать и в соответствующих месяцу и дню частях выходной строки. Приглашаем вас поэкспериментировать с прочими форматами входных дат и посмотреть, как изменится матрица внимания.

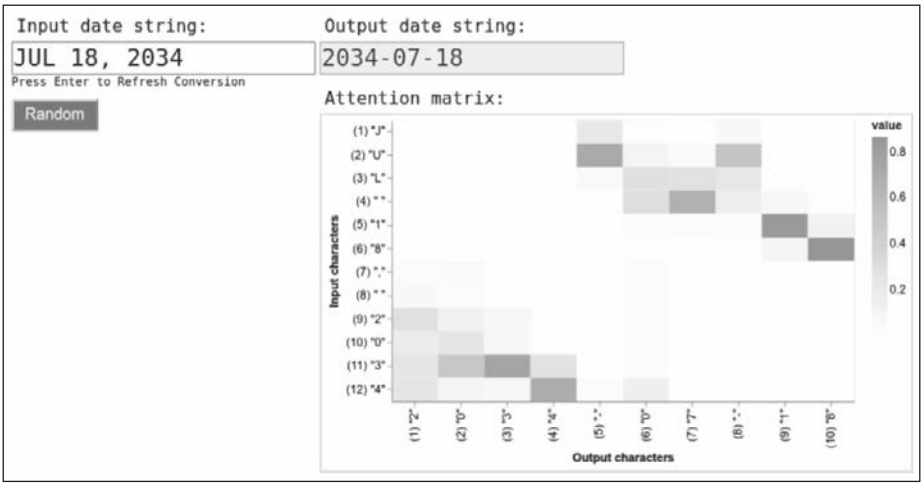


Рис. 9.9. Кодировщик — декодировщик для преобразования дат, основанный на механизме внимания. Внизу справа отображена матрица внимания для конкретной пары входного/выходного сигнала

9.3.2. Архитектура «кодировщик — декодировщик» и механизм внимания

Это раздел поможет вам прочувствовать, как решаются задачи seq2seq с помощью архитектуры «кодировщик — декодировщик» и какую роль играет в ней механизм внимания. В следующем разделе эти механизмы обсуждаются во всех подробностях и приводится соответствующий код.

До сих пор выходной сигнал всех приведенных в этой книге нейронных сетей представлял собой одиночный элемент. Для сети регрессии выходной сигнал представлял собой одно число, для сети классификации — одно распределение вероятности по нескольким возможным категориям. Задача же преобразования дат, которую мы решаем, от них отличается: нам нужно предсказать не один элемент, а несколько. А именно, требуется предсказать ровно десять символов формата даты ISO-8601. Как сделать это с помощью нейронной сети?

Решение таково: создать сеть, на выходе которой — последовательность элементов. В частности, поскольку выходная последовательность состоит из дискретных символов «алфавита», включающего ровно 11 элементов (цифры с 0 до 9, а также дефис), форма выходного тензора сети может быть трехмерной: `[numExamples, OUTPUT_LENGTH, OUTPUT_VOCAB_SIZE]`. Первое измерение (`numExamples`) — обычное измерение примеров данных, позволяющее обрабатывать данные по батчам, как и во всех прочих сетях в этой книге. `OUTPUT_LENGTH` равно `10`, это фиксированная длина выходного строкового значения для даты в формате ISO-8601. `OUTPUT_VOCAB_SIZE` — длина выходного словаря (точнее, выходного «алфавита»), включающего цифры от 0 до 9 и символ дефиса (`-`), помимо нескольких специальных символов, которые мы обсудим далее.

Вот и все о выходном сигнале модели. А как насчет входных данных? Оказывается, модель принимает на входе *два* сигнала вместо одного. Нашу модель можно разделить на две части, кодировщик и декодировщик, как показано схематически на рис. 9.10. Первый входной сигнал модели поступает в кодировщик и представляет собой саму входную строку с датой, имеющую вид последовательности индексов символов в форме `[numExamples, INPUT_LENGTH]`. `INPUT_LENGTH` — максимально возможная длина среди всех поддерживаемых форматов дат (равная, как оказывается, 12). Входные данные, которые меньше этой длины, дополняются нулями с конца. Второй входной сигнал поступает в декодировщик и представляет собой результат преобразования, сдвинутый вправо на один временной шаг, в форме `[numExamples, OUTPUT_LENGTH]`.

Погодите-ка, смысл первого входного сигнала понятен — это входная строка с датой, но зачем модель принимает в качестве дополнительного входного сигнала еще и результат преобразования? Разве он не должен быть *выходным сигналом* модели? Все дело в сдвиге по времени результата преобразования. Учтите, что второй входной сигнал — *не совсем* результат преобразования, а его сдвинутая по времени ровно на один временной шаг версия. Например, если желаемым результатом

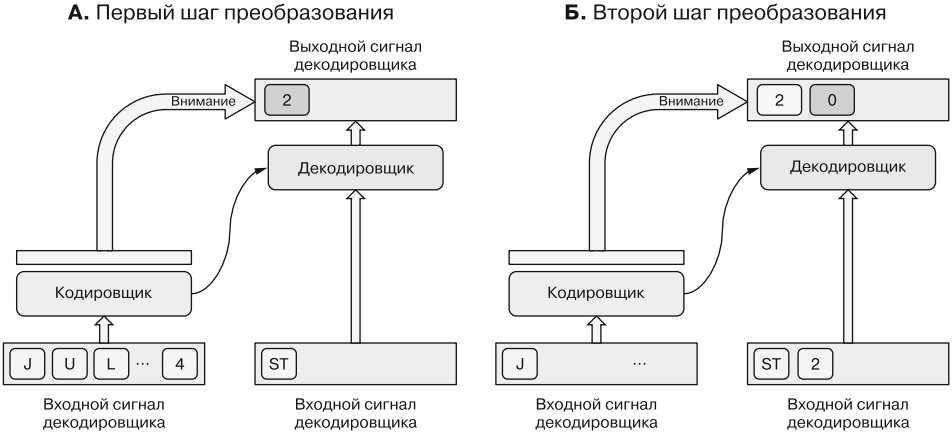


Рис. 9.10. Преобразование входного строкового значения с датой в выходное в архитектуре типа «кодировщик — декодировщик». ST — специальный токен начала последовательности для входного и выходного сигналов декодировщика. Блоки А и В демонстрируют первый и второй шаги преобразования соответственно. На первом шаге преобразования генерируется первый символ выходного сигнала (2). В ходе второго шага генерируется второй символ (0). Остальные шаги соответствуют этому же шаблону, поэтому мы их опустим

преобразования во время обучения было `"2034-07-18"`, то вторым входным сигналом модели будет `"<ST>2034-07-1"`, где `<ST>` — специальный символ начала последовательности. Благодаря этому сдвигу входного сигнала декодировщик знает о том, какая выходная последовательность уже сгенерирована. Это упрощает отслеживание декодировщиком хода преобразования.

Этот процесс напоминает человеческую речь. При выражении мысли в словах основные умственные усилия затрачиваются на две вещи: саму выражаемую идею и отслеживание уже сказанного. Вторая часть очень важна для обеспечения связности и полноты речи, а также отсутствия повторов. Наша модель функционирует таким же образом: для генерации каждого из выходных символов используется информация как о входной строке с датой, так и об уже сгенерированных выходных символах.

Сдвиг по времени результата преобразования возможен на этапе обучения, поскольку мы заранее знаем правильный результат преобразования. Но как это возможно во время выполнения вывода? Ответ на этот вопрос понятен из двух блоков, изображенных на рис. 9.10: мы генерируем выходные символы по одному[1]. Как демонстрирует блок А, сначала мы помещаем символ `ST` в начало входного сигнала декодировщика. Один шаг вывода (один вызов `Model.predict()`) дает один новый элемент выходного сигнала (`2` в блоке А). После этого новый выходной элемент добавляется в конец входной последовательности декодировщика. Далее переходим

[1] Этот алгоритм пошагового преобразования реализует функция runSeq2SeqInference() из файла date-conversion-attention/model.js.

к следующему шагу преобразования. Модель видит только что сгенерированный выходной символ `2` во входной последовательности декодировщика (блок B). На этом шаге производится еще один вызов `Model.predict()` и генерируется новый выходной символ (`0`), который тоже добавляется в конец входной последовательности декодировщика. Этот процесс повторяется до тех пор, пока не будет получена желаемая длина выходной последовательности (в данном случае 10). Учтите, что выходная последовательность не включает символ `ST`, так что может служить непосредственно выходным сигналом всего алгоритма.

Роль механизма внимания

Задача механизма внимания состоит в том, чтобы каждый выходной символ уделял внимание соответствующим символам входной последовательности. Например, часть `"7"` выходной строки `"2034-07-18"` должна уделять внимание части `"JUL"` входной строки с датой. Это тоже аналогично генерации речи у людей. Например, при переводе предложения с языка A на язык B каждое слово выходного предложения обычно определяется небольшим числом слов из входного предложения.

Кажется, что это очевидно, сложно представить себе лучший подход. Но изобретение механизма внимания исследователями в области глубокого обучения в 2014–2015 годах было крупнейшим достижением в этой сфере. Чтобы понять причины этого, взгляните на стрелку, соединяющую прямоугольник *«Кодировщик»* с прямоугольником *«Декодировщик»* в блоке A на рис. 9.10. Эта стрелка соответствует последнему выходному сигналу LSTM в кодирующей части модели, передаваемому LSTM в декодирующей части модели в качестве начального состояния. Напомним, что начальное состояние RNN обычно состоит из нулей (как, например, в simpleRNN в разделе 9.1.2), однако TensorFlow.js дает возможность задавать начальное состояние RNN равным любому тензору подходящей формы. Этой возможностью можно воспользоваться для передачи информации из расположенных ранее частей конвейера в LSTM. В данном случае этот механизм задействуется соединением «кодировщик — декодировщик» для предоставления LSTM-слою декодировщика доступа к кодированной входной последовательности.

Однако начальное состояние представляет собой всю входную последовательность, упакованную в один вектор. Такое представление оказывается слишком сжатым для распаковки декодировщиком, особенно если последовательности более длинные и сложные, такие как предложения в типичных задачах машинного перевода. Тут-то и оказывается полезным механизм внимания.

Механизм внимания расширяет поле зрения декодировщика. Механизм внимания обращается ко всей последовательности выходного сигнала кодировщика, а не только к его итоговому выходному сигналу. На каждом шаге преобразования этот механизм уделяет внимание определенным временным шагам выходной последовательности кодировщика, чтобы определить, какой выходной символ генерировать. Например, первый шаг преобразования может уделять внимание первым двум входным символам, а второй — второму и третьему входным символам и т. д

(конкретный пример подобной матрицы внимания приведен на рис. 9.10). Как и все весовые параметры нейронной сети, модель внимания *обучается* уделять внимание нужным вещам, вместо того чтобы жестко «зашивать» стратегию работы в коде. В результате модель оказывается гибкой и обладает большими возможностями: она может обучиться уделять внимание различным частям входной последовательности в зависимости от самой входной последовательности и сгенерированной на этот момент части выходной последовательности.

Дальнейшее обсуждение механизма «кодировщик — декодировщик» бессмысленно без анализа кода или открытия черных ящиков кодировщика и декодировщика, а также механизма внимания. Если подобное обсуждение в общих чертах для вас недостаточно конкретно, прочитайте следующий раздел, в котором мы углубимся в детали модели. Если вы хотите лучше разобраться в основанной на механизме внимания архитектуре «кодировщик — декодировщик», затраченные на это усилия себя оправдают. Чтобы побудить вас его прочитать, скажем лишь, что эта же архитектура лежит, в частности, в основе самых современных моделей машинного перевода (например, Google Neural Machine Translation, GNMT), хотя число слоев LSTM в этих промышленных моделях намного больше и они обучаются на намного больших массивах данных, чем наша простая модель преобразования дат.

9.3.3. Заглянем глубже в модель «кодировщик — декодировщик», основанную на механизме внимания

На рис. 9.11 показаны подробности внутреннего устройства прямоугольников с рис. 9.10. Для наглядности лучше смотреть на него вместе с кодом создания модели — функцией `createModel()` из файла `date-conversion-attention/model.js`. Далее рассмотрим наиболее важные аспекты этого кода.

Во-первых, определяем несколько констант для слоев вложений и LSTM в кодировщике и декодировщике:

```
const embeddingDims = 64;
const lstmUnits = 64;
```

Формируемая модель принимает два входных сигнала, так что нам придется воспользоваться функциональным API вместо последовательного API. Начнем с символических входных сигналов модели для входного сигнала кодировщика и декодировщика соответственно:

```
const encoderInput = tf.input({shape: [inputLength]});
const decoderInput = tf.input({shape: [outputLength]});
```

Как кодировщик, так и декодировщик применяют к своим входным последовательностям слой вложений. Соответствующий код для кодировщика выглядит следующим образом:

```
let encoder = tf.layers.embedding({
  inputDim: inputVocabSize,
  outputDim: embeddingDims,
  inputLength,
  maskZero: true
}).apply(encoderInput);
```

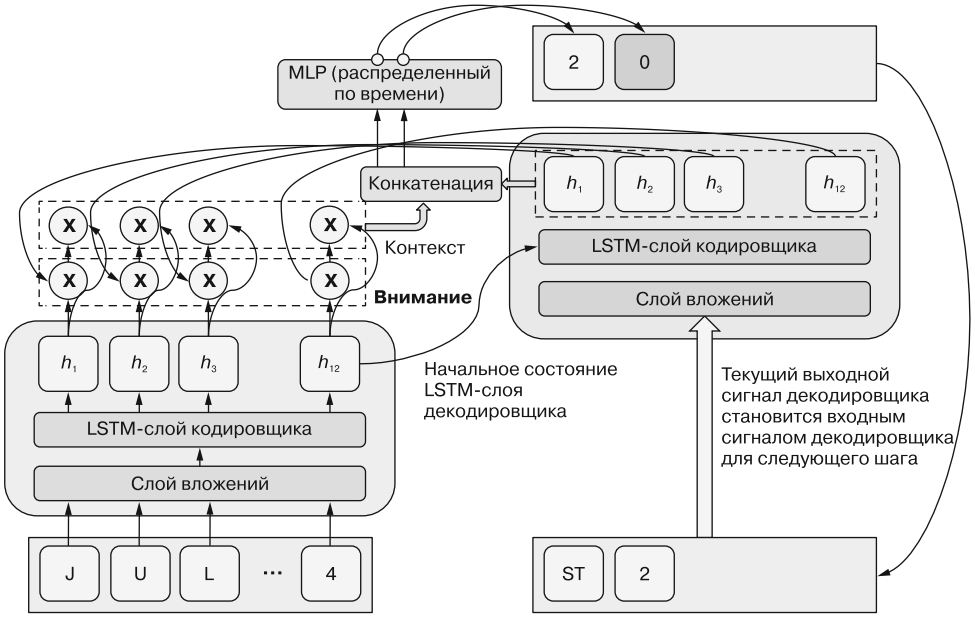


Рис. 9.11. Заглядываем глубже в основанную на механизме внимания модель «кодировщик — декодировщик». Этот рисунок можно считать расширенной, с дополнительными подробностями, версией архитектуры «кодировщик — декодировщик» с рис. 9.10

Все так же, как и в слоях вложений для задачи анализа тональностей IMDb, только речь идет о вложениях символов, а не слов. А значит, сфера применения метода вложений не ограничивается словами. На самом деле он достаточно гибок для применения к любому конечному дискретному множеству, например к музыкальным жанрам, статьям на новостном веб-сайте, аэропортам страны и т. д. Настройка `maskZero: true` слоя вложений предписывает расположенному далее по конвейеру LSTM-слою пропускать шаги, где все значения — нулевые, и таким образом экономит вычислительные ресурсы на уже завершившихся последовательностях.

LSTM — тип RNN, который мы пока что подробно не рассматривали. Не станем углубляться в нюансы его внутренней структуры. Достаточно сказать, что он подобен GRU (см. рис. 9.4) в том, что решает проблему исчезающего градиента за счет упрощения переноса состояния через несколько временны*х* шагов. В посте в блоге Криса Олаха (Chris Olah) *Understanding LSTM Networks* («Разбираемся с LSTM-сетями»), ссылку на которое вы можете найти в конце главы, приведены прекрасный

обзор и визуализация структуры и механизмов LSTM. LSTM-слой нашего кодировщика применяется к векторам вложений символов:

```
encoder = tf.layers.lstm({
  units: lstmUnits,
  returnSequences: true
}).apply(encoder);
```

Благодаря настройке `returnSequences: true` выходной сигнал LSTM может представлять собой последовательность выходных векторов, а не только одного выходного вектора (по умолчанию) — итогового выходного сигнала, как это было в моделях предсказания температуры и анализа тональностей. Этот шаг необходим для расположенного далее по конвейеру механизма внимания.

Слой `GetLastTimestepLayer`, следующий за слоем LSTM кодировщика, — нестандартный:

```
const encoderLast = new GetLastTimestepLayer({
  name: 'encoderLast'
}).apply(encoder);
```

Он просто производит срез тензора «время — последовательность» по измерению времени (второму измерению) и выдает на выходе последний временной шаг. Благодаря этому мы можем отправить итоговое состояние LSTM-слоя кодировщика на вход LSTM-слоя декодировщика в качестве начального состояния. Эта связь — один из способов получения декодировщиком информации о входной последовательности. Она показана на рис. 9.11 с помощью стрелки, соединяющей *h12* в зеленом прямоугольнике кодировщика со слоем LSTM декодировщика в голубом прямоугольнике кодировщика.

Относящаяся к декодировщику часть кода начинается со слоя вложений и LSTM-слоя, напоминающих топологию кодировщика:

```
let decoder = tf.layers.embedding({
  inputDim: outputVocabSize,
  outputDim: embeddingDims,
  inputLength: outputLength,
  maskZero: true
}).apply(decoderInput);
decoder = tf.layers.lstm({
  units: lstmUnits,
  returnSequences: true
}).apply(decoder, {initialState: [encoderLast, encoderLast]});
```

Обратите внимание, как в последней строке этого фрагмента кода итоговое состояние кодировщика используется в качестве начального состояния декодировщика. Не удивляйтесь повторению символического тензора `encoderLast` в последней строке кода: дело в том, что LSTM-слой содержит два состояния, в отличие от структуры с одним состоянием, как в simpleRNN и GRU.

Дополнительный, еще более многообещающий способ взглянуть на входные последовательности для декодировщика — конечно, механизм внимания. Внима-

ние — это скалярное произведение (поэлементное) выходного сигнала LSTM-слоя кодировщика и выходного сигнала LSTM-слоя декодировщика с последующей многомерной логистической функцией активации:

```
let attention = tf.layers.dot({axes: [2, 2]}).apply([decoder, encoder]);
attention = tf.layers.activation({
  activation: 'softmax',
  name: 'attention'
}).apply(attention);
```

Форма выходного сигнала LSTM-слоя кодировщика — `[null, 12, 64]`, где `12` — длина входной последовательности, а `64` — размер LSTM-слоя. Форма выходного сигнала LSTM-слоя декодировщика — `[null, 10, 64]`, где `10` — длина выходной последовательности, а `64` — размер LSTM-слоя. Скалярное произведение между ними определяется по последнему измерению (измерению LSTM-признаков), и получается форма `[null, 10, 12]` (то есть `[null, inputLength, outputLength]`). В результате применения к скалярному произведению многомерной логистической функции активации значения превращаются в оценки вероятности, которые заведомо неотрицательны и в сумме по каждому столбцу матрицы равны 1. Эта матрица внимания играет главную роль в нашей модели. Именно ее значение было визуализировано на рис. 9.9.

Далее матрица внимания применяется к выходной последовательности LSTM-слоя кодировщика, посредством чего процесс преобразования обучается уделять внимание различным элементам входной последовательности (в закодированной форме) на каждом из шагов. Результат применения механизма внимания к входному сигналу кодировщика называется *контекстом* (context):

```
const context = tf.layers.dot({
  axes: [2, 1],
  name: 'context'
}).apply([attention, encoder]);
```

Форма контекста — `[null, 10, 64]` (то есть `[null, outputLength, lstmUnits]`). Он склеивается с выходным сигналом декодировщика, также формы `[null, 10, 64]`. Таким образом, форма результата конкатенации будет `[null, 10, 128]`:

```
const decoderCombinedContext =
    tf.layers.concatenate().apply([context, decoder]);
```

`decoderCombinedContext` содержит векторы признаков, попадающие в последний этап модели, на котором генерируются выходные символы.

Эти выходные символы генерируются с помощью MLP, включающего один скрытый слой и выходной слой с многомерной логистической функцией:

```
let output = tf.layers.timeDistributed({
  layer: tf.layers.dense({
    units: lstmUnits,
    activation: 'tanh'
  })
}).apply(decoderCombinedContext);
output = tf.layers.timeDistributed({
```

```
  layer: tf.layers.dense({
    units: outputVocabSize,
    activation: 'softmax'
  })
}).apply(output);
```

Благодаря слою `timeDistributed` на всех шагах используется один и тот же MLP. Слой `timeDistributed` принимает на входе слой и многократно вызывает его для всех шагов по измерению времени (то есть второму измерению) своего входного сигнала. В результате форма входных признаков `[null, 10, 128]` преобразуется в `[null, 10, 13]`, где `13` соответствует 11 возможным символам формата ISO-8601, а также двум специальным символам — дополняющему символу и символу начала последовательности.

Далее мы собираем все эти составные части воедино в объект `tf.Model` с двумя входными сигналами и одним выходным:

```
const model = tf.model({
  inputs: [encoderInput, decoderInput],
  outputs: output
});
```

Для подготовки к обучению вызываем метод `compile()`, указывая категориальную перекрестную энтропию в качестве функции потерь. Мы выбрали именно эту функцию потерь, поскольку наша задача преобразования, по сути, является задачей классификации — на каждом временном шаге выбирается один символ из множества всех возможных символов:

```
model.compile({
  loss: 'categoricalCrossentropy',
  optimizer: 'adam'
});
```

При выводе для получения выходного символа-«победителя» к выходному тензору модели применяется функция `argMax()`. На каждом шаге преобразования выходной символ-«победитель» присоединяется к концу выходной последовательности декодировщика для использования на следующем шаге преобразования (см. крайнюю справа стрелку на рис. 9.11). Как мы уже упоминали, в результате этого процесса преобразования постепенно формируется вся выходная последовательность.

Материалы для дальнейшего изучения

- *Olah C.* Understanding LSTM Networks // blog, 27 Aug. 2015: http://mng.bz/m4Wa.
- *Olah C., Carter S.* Attention and Augmented Recurrent Neural Networks // Distill, 8 Sept. 2016: https://distill.pub/2016/augmented-rnns/.
- *Karpathy A.* The Unreasonable Effectiveness of Recurrent Neural Networks // blog, 21 May 2015: http://mng.bz/6wK6.

- *Ahmed Z.* How to Visualize Your Recurrent Neural Network with Attention in Keras // Medium, 29 June 2017: http://mng.bz/6w2e.
- В примере преобразования дат мы описали методику декодирования на основе функции `argMax()`. Такой подход часто называют методикой *«жадного» декодирования* (greedy decoding), поскольку на каждом шаге выделяется выходной символ с максимальной вероятностью. Одним из популярных альтернативных подходов является *декодирование с поиском по лучу* (beam-search decoding), в котором исследуется более широкий диапазон возможных выходных последовательностей для определения наилучшей. Прочитать об этом методе можно здесь: *Brownlee Jason.* How to Implement a Beam Search Decoder for Natural Language Processing, 5 Jan. 2018: https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/.
- *Raaijmakers S.* Deep Learning for Natural Language Processing, Manning Publications: www.manning.com/books/deep-learning-for-natural-language-processing.

Упражнения

1. Попробуйте поменять порядок элементов данных для различных непоследовательных данных. Убедитесь, что подобная перестановка не влияет на значения потерь и показатели (например, безошибочность) моделирования (не считая случайных колебаний вследствие инициализации весовых параметров случайными значениями). Сделайте это для следующих двух задач.
   - В примере ирисов Фишера (из главы 3) поменяйте порядок четырех числовых признаков (длина чашелистиков, ширина чашелистиков, длина лепестков и ширина лепестков), внеся изменения в строку:

     ```
     shuffledData.push(data[indices[i]]);
     ```

     в файле `iris/data.js` репозитория tfjs-examples. В частности, измените порядок упомянутых четырех элементов в `data[indices[i]]`. Сделать это можно с помощью вызовов методов `slice()` и `concat()` JavaScript-массива. Учтите, что менять порядок следует одинаково для всех примеров данных. Можете написать JavaScript-функцию для переупорядочения.
   - Попробуйте переупорядочить 240 временны*х* шагов *и* 14 числовых признаков (показаний метеорологических приборов) для линейного регрессора и MLP, разработанных нами для задачи Jena-weather. Если точнее, для этого можно внести изменения в функцию `nextBatchFn()` из файла `jena-weather/data.js`. Проще всего реализовать переупорядочение — в строке:

     ```
     samples.set(value, j, exampleRow, exampleCol++);
     ```

     где можно изменить индекс `exampleRow` на новое значение с помощью функции, выполняющей фиксированную перестановку элементов, и аналогичным образом изменить `exampleCol`.

2. Созданная для анализа тональностей IMDb одномерная сверточная сеть состояла всего из одного слоя conv1d (см. листинг 9.8). Как мы обсуждали, добавление поверх него дополнительных слоев conv1d дает более глубокую одномерную сверточную сеть, способную захватывать информацию об упорядоченности на бóльших диапазонах слов. В этом упражнении вам предстоит модифицировать код функции `buildModel()` из файла `sentiment/train.js`. Задача: добавить поверх уже существующего еще один слой conv1d, заново обучить модель и посмотреть, повысится ли степень безошибочности классификации. Число фильтров и размер ядра нового слоя можно взять такие же, как у уже существующего. Взгляните также на формы выходных сигналов в сводке топологии модифицированной модели и убедитесь, что понимаете, почему подобные параметры `filters` и `kernelSize` обусловливают такую форму выходного тензора нового слоя conv1d.
3. Попробуйте добавить в пример date-conversion-attention несколько дополнительных входных форматов дат. Можете выбрать новые форматы из приведенных далее, отсортированных в порядке возрастания сложности написания кода. А можете придумать собственные форматы дат.
   - Формат YYYY-MMM-DD, например, `"2012-MAR-08"` или `"2012-MAR-18"`. В зависимости от того, дополняются ли спереди нулем обозначаемые одной цифрой дни месяца (как в 12/03/2015), он может представлять собой два различных формата. Впрочем, независимо от дополнения максимальная длина этого формата меньше 12, а все возможные символы уже входят в словарь `INPUT_VOCAB` в файле `date-conversion-attention/date_format.js`. Следовательно, достаточно добавить функцию или две в этот файл, как в уже существующих, например, `dateTupleToMMMSpaceDDSpaceYY()`. Не забудьте добавить новую (-ые) функцию (-и) в массив `INPUT_FNS` в файле, чтобы включить их в процесс обучения. Рекомендуем также добавлять модульные тесты для новых функций форматирования дат в файл `date-conversion-attention/date_format_test.js`.
   - Формат, в котором день месяца обозначается порядковым числительным, например Mar 8th, 2012. Это тот же формат, что и уже существующий `dateTupleToMMMSpaceDDCommaSpaceYYYY()`, только ко дням месяца добавляются порядковые суффиксы (`"st"`, `"nd"` и `"th"`). Ваша новая функция должна определять суффикс по значению дня. Кроме того, вам нужно пересмотреть значение константы `INPUT_LENGTH` из файла `date_format_test.js` в бóльшую сторону, поскольку максимально возможная длина строки даты в этом формате превышает текущее значение, равное 12. Помимо этого, необходимо добавить в словарь буквы `"t"` и `"h"`, которые не встречаются в трехбуквенных названиях месяцев.
   - Возьмем теперь формат с полным английским названием месяца наподобие March 8th, 2012. Какова максимально возможная длина входной строки с датой? Какие соответствующие изменения необходимо внести в словарь `INPUT_VOCAB` из файла `date_format.js`?

Резюме

- Благодаря возможности выделять и усваивать заключенную в порядке элементов информацию RNN способны на большее, чем модели прямого распространения (например, MLP), при решении задач, связанных с обработкой последовательных входных данных. Мы показали это на примере применения simpleRNN и GRU к задаче предсказания температуры.
- В TensorFlow.js доступны три типа слоев RNN: simpleRNN, GRU и LSTM. Последние два типа сложнее simpleRNN и используют более сложную внутреннюю структуру, чтобы обеспечить передачу информации о состоянии на большое число временны́х шагов с целью устранения проблемы исчезающего градиента. GRU требуют меньшего объема вычислений, чем LSTM. На практике в большинстве задач, вероятно, имеет смысл применять GRU и LSTM.
- При создании нейронных сетей для обработки текста необходимо сначала представить текстовые входные данные в виде числовых векторов. Этот процесс называется векторизацией текста. Наиболее широко распространенные методы векторизации текста — унитарное и федеративное кодирование, а также обладающий бóльшими возможностями метод вложений слов.
- В методе вложений слов каждое слово представляется в виде неразреженного вектора, значения элементов которого усваиваются сетью в ходе обратного распространения ошибки точно так же, как и все прочие весовые параметры нейронной сети. Для выполнения вложений слов в TensorFlow.js служит функция `tf.layers.embedding()`.
- Задачи преобразования последовательностей в последовательности (seq2seq) отличаются от задач регрессии и классификации на основе последовательностей тем, что в качестве выходного сигнала генерируется новая последовательность. При создании предназначенной для решения задач seq2seq архитектуры «кодировщик — декодировщик» могут использоваться RNN (наряду с прочими типами слоев).
- В задачах seq2seq благодаря механизму внимания различные элементы выходной последовательности могут выборочно зависеть от определенных элементов входной последовательности. Мы показали, как обучить сеть типа «кодировщик — декодировщик» на основе механизма внимания для решения простой задачи преобразования дат и визуализировать матрицу внимания во время выполнения вывода.

10 Генеративное глубокое обучение

В этой главе

- Что такое генеративное глубокое обучение, области его использования и отличия от обсуждавшихся ранее задач глубокого обучения.
- Генерация текста с помощью RNN.
- Латентное пространство как основа для генерации новых изображений на примере вариационного автокодировщика.
- Основы генеративных состязательных сетей (GAN).

Среди наиболее впечатляющих результатов работы глубоких нейронных сетей — генерация правдоподобно выглядящих/звучащих изображений и звуков. В настоящее время глубокие нейронные сети способны создавать чрезвычайно реалистичные изображения человеческих лиц[1], синтезировать естественно звучащую речь[2], писать вполне связные тексты[3], и это далеко не все. Подобные *генеративные* (generative)[4]

---

1 *Karras T., Laine S., Aila T.* A Style-Based Generator Architecture for Generative Adversarial Networks // submitted 12 Dec. 2018. https://arxiv.org/abs/1812.04948. См. демонстрацию по адресу https://thispersondoesnotexist.com/.

2 *Oord A. van den, Dieleman S.* WaveNet: A Generative Model for Raw Audio // blog, 8 Sept. 2016. http://mng.bz/MOrn.

3 Better Language Models and Their Implications // OpenAI, 2019. https://openai.com/blog/better-language-models/.

4 В русскоязычной литературе часто называются также «порождающими». — *Примеч. пер.*

модели могут пригодиться для множества целей, включая помощь в художественном творчестве, модификацию существующего контента в зависимости от определенных условий и дополнение существующих наборов данных для решения других задач глубокого обучения[1].

Помимо чисто практических приложений, например наведения лоска на селфи потенциального покупателя косметики, генеративные модели заслуживают внимания и по теоретическим причинам. Генеративное и дискриминативное моделирование — два принципиально различных типа моделей в машинном обучении. Все встречавшиеся нам до сих пор в книге модели были *дискриминативными* (discriminative)[2]. Подобные модели предназначены для отображения входного сигнала в дискретное или непрерывное значение, вне зависимости от того, в ходе какого процесса этот входной сигнал был сгенерирован. Вспомните созданные нами классификаторы фишинговых сайтов, ирисов Фишера, цифр из набора MNIST и звуков речи, а также регрессор для цен на недвижимость. Генеративные модели, напротив, предназначены для математической имитации процесса генерации примеров данных различных классов. Но после усвоения генеративной моделью способа генерации она способна решать и дискриминативные задачи. Таким образом, можно считать, что генеративные модели лучше «понимают» данные, по сравнению с дискриминативными.

В этой главе мы рассмотрим основы глубоких генеративных моделей, предназначенных для текста и изображений. К концу главы вы узнаете, какие идеи лежат в основе использующих RNN моделей языка, автокодировщиков, ориентированных на обработку изображений, и генеративных состязательных сетей. Вы также познакомитесь с паттерном реализации подобных моделей в TensorFlow.js и при необходимости сможете применить их к своим наборам данных.

10.1. Генерация текста с помощью LSTM

Начнем с генерации текста. Для этого воспользуемся RNN, с которыми познакомили вас в предыдущей главе. И хотя описанная далее методика применяется здесь для генерации текста, она вовсе не ограничивается только этой предметной областью. Ее можно приспособить и для генерации прочих типов последовательностей, например музыкальных произведений, — достаточно обеспечить возможность представления музыкальных нот подходящим образом и найти подходящий обучающий набор данных[3]. Аналогичным образом можно приспособить их для генерации росчерков

---

[1] *Antoniou A., Storkey A., Edwards H.* Data Augmentation Generative Adversarial Networks // submitted 12 Nov. 2017. https://arxiv.org/abs/1711.04340.

[2] В русскоязычной литературе иногда встречается название «различающие модели». — *Примеч. пер.*

[3] Например, см. модель Performance-RNN из проекта Magenta компании Google: https://magenta.tensorflow.org/performance-rnn.

пера при рисовании, создавая прекрасные наброски[1] или даже реалистично выглядящие японские иероглифы[2].

10.1.1. Предсказание следующего символа: простой способ генерации текста

Во-первых, давайте сформулируем задачу генерации текста. Пусть в качестве входных обучающих данных дан корпус текстовых данных достаточного размера (хотя бы несколько мегабайт), например полное собрание сочинений Шекспира (в виде очень длинной строки). Необходимо обучить модель генерировать новые тексты, как можно более *похожие* на обучающие данные. Ключевое слово здесь, конечно, «похожие». Пока не будем утруждать себя точным определением этого слова. Его смысл прояснится после демонстрации метода и результатов его работы.

Попробуем сформулировать задачу в парадигме глубокого обучения. В рассмотренном в предыдущей главе примере преобразования форматов дат мы показали, как сгенерировать четко отформатированную выходную последовательность на основе небрежно отформатированной входной. У этой задачи преобразования текста в текст было четко заданное решение: правильная строка с датой в формате ISO-8601. Однако задача генерации текста этим требованиям не отвечает. Никакой явной входной последовательности нет, да и «правильный» выходной сигнал четко не определен: просто требуется сгенерировать нечто «правдоподобно выглядящее». Как же нам поступить?

Решение состоит в создании модели, которая бы предсказывала, какой символ следует за заданной последовательностью символов. Это называется *предсказанием следующего символа* (next-character prediction). Например, получив в качестве входной последовательности строку *Love looks not with the eyes, b*, модель, хорошо обученная на наборе данных текстов Шекспира, должна с высокой степенью вероятности предсказать символ *u*. Впрочем, она предсказывает только один символ. Как же сгенерировать с ее помощью последовательность символов? Очень просто: формируем новую входную последовательность той же длины, что и раньше, сдвигая предыдущую входную последовательность на один символ влево, отбрасывая первый символ и вставляя в конец последовательности только что сгенерированный символ *(u)*. В результате получаем новую входную последовательность для алгоритма предсказания следующего символа, а именно, *ove looks not with the eyes, bu*. По такой входной последовательности модель должна с высокой степенью вероятности предсказать символ *t*. Этот процесс, продемонстрированный на рис. 10.1, можно повторять столько раз, сколько требуется, для генерации последовательности нужной длины. Конечно, при этом требуется начальный фрагмент текста в качестве отправной точки, который можно просто выбрать случайным образом из корпуса текста.

---

[1] Например, см. модель Sketch-RNN, созданную Дэвидом Ха (David Ha) и Дугласом Экком (Douglas Eck): http://mng.bz/omyv.

[2] *Ha D.* Recurrent Net Dreams Up Fake Chinese Characters in Vector Format with TensorFlow // blog, 28 Dec. 2015. http://mng.bz/nvX4.

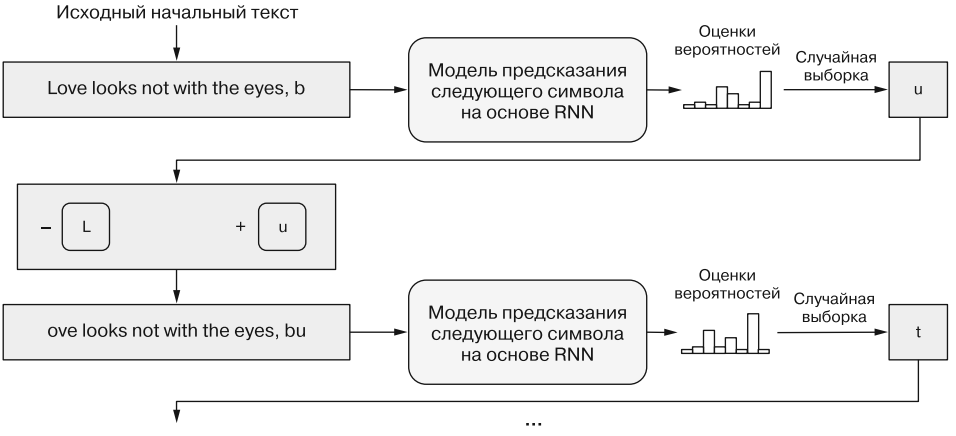


Рис. 10.1. Схематическая иллюстрация возможностей генерации текстовой последовательности по входному фрагменту текста в качестве начального значения с помощью модели предсказания очередного символа на основе RNN. На каждом шаге RNN предсказывает следующий символ по входному тексту. Далее производится конкатенация входного текста с предсказанным следующим символом, а первый символ текста отбрасывается. Полученный результат служит входным сигналом для очередного шага. На каждом шаге RNN выдает оценки вероятностей всех возможных символов из множества. Фактически следующий символ определяется на основе случайной выборки

Подобная формулировка превращает задачу генерации последовательности в задачу классификации на основе последовательности, аналогичную задаче анализа тональностей обзоров из IMDb в главе 9, в которой предсказывался бинарный класс по входному сигналу фиксированной длины. Модель генерации текста, по существу, делает то же самое, только речь идет о многоклассовой классификации с N возможными классами, где N — размер множества символов, а именно, количество всех уникальных символов в наборе текстов.

Подобная формулировка предсказания следующего символа далеко не нова в истории обработки естественного языка и теории вычислительной техники. Клод Шеннон, один из пионеров теории информации, провел эксперимент, в котором участников просили угадать следующую букву по короткому фрагменту текста на английском языке[1]. Благодаря этому эксперименту Шеннон смог оценить среднюю степень неопределенности каждой буквы типичного текста на английском языке для заданного контекста. Эта неопределенность, оказавшаяся равной 1,3 бита энтропии, отражает, какой в среднем объем информации несет каждая английская буква.

Результат в 1,3 бита меньше, чем неопределенность при совершенно случайном наборе из 26 символов: $\log_2(26) = 4{,}7$ бита. Интуитивно это понятно, поскольку в английском языке порядок букв случайным не бывает, буквы следуют определенным закономерностям. Если рассматривать на самом низком уровне, только определенные последовательности букв являются допустимыми английскими словами.

[1] Первоисточник — статью 1951 года — можно найти по адресу http://mng.bz/5AzB.

На более высоком уровне, только слова в определенном порядке удовлетворяют требованиям английской грамматики. А на еще более высоком уровне лишь небольшое подмножество грамматически правильных предложений является осмысленным.

Если задуматься, именно в этом и состоит наша задача генерации текста: усвоить закономерности на всех уровнях. По существу, наша модель обучается делать именно то, что делали участники исследования Шеннона, — угадывать следующий символ. А теперь взглянем на код этого примера и обсудим, как он работает. Запомните результат Шеннона (1,3 бита), поскольку мы еще вернемся к нему.

10.1.2. Пример lstm-text-generation

Пример lstm-text-generation из репозитория tfjs-examples включает обучение основанной на LSTM модели предсказания следующего символа и генерацию с ее помощью нового текста. Оба этапа — обучения и генерации — выполняются на JavaScript с помощью TensorFlow.js. Можете запустить этот пример как в браузере, так и в среде прикладной части с помощью Node.js. Первый вариант отличается более наглядным и интерактивным интерфейсом, но скорость обучения во втором — выше.

Для выполнения примера в браузере выполните следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/lstm-text-generation
yarn && yarn watch
```

На открывшейся веб-странице можно выбрать и загрузить для обучения модели один из четырех предлагаемых текстовых наборов данных. В дальнейшем обсуждении используется набор данных текстов Шекспира. После загрузки данных можно создать для них модель, нажав кнопку Create Model (Создать модель). В текстовом поле ввода можно указать количество нейронов создаваемого LSTM-слоя. По умолчанию оно равно 128, но вы можете поэкспериментировать с другими значениями, например 64. Если ввести несколько чисел, разделенных запятыми (например, `128,128`), будет создана модель с несколькими LSTM-слоями, один поверх другого.

Для обучения в прикладной части с помощью модулей tfjs-node или tfjs-node-gpu воспользуйтесь командой `yarn train` вместо `yarn watch`:

```
yarn train shakespeare \
      --lstmLayerSize 128,128 \
      --epochs 120 \
      --savePath ./my-shakespeare-model
```

При наличии настроенного должным образом GPU с поддержкой CUDA можете добавить в эту команду флаг `--gpu`, чтобы выполнять обучение на GPU с существенно более высокой скоростью. Флаг `--lstmLayerSize` играет ту же роль, что и текстовое поле, куда нужно ввести размер LSTM в браузерной версии примера. Приведенная выше команда создает и обучает модель, состоящую из двух LSTM-слоев, каждый с 128 нейронами, размещенными один поверх другого.

У обучаемой здесь модели многоярусная LSTM-архитектура. Что означает размещение LSTM-слоев *друг поверх друга* (stacking)? По существу, подобно тому, как

размещение друг над другом плотных слоев в MLP расширяет разрешающие возможности модели, так и размещение друг над другом нескольких LSTM-слоев дает возможность многоэтапного преобразования представлений seq2seq входной последовательности, перед тем как она будет преобразована в завершающем LSTM-слое в итоговый результат регрессии или классификации. Данная архитектура схематически изображена на рис. 10.2. Важно отметить, что свойство `returnSequence` первого LSTM-слоя равно `true`, а потому он генерирует выходную последовательность, включающую выходной сигнал для всех отдельных элементов входной последовательности. Это позволяет подать выходной сигнал первого LSTM-слоя на вход второго, ведь LSTM-слой ожидает последовательные входные данные, а не входной сигнал из одного элемента.

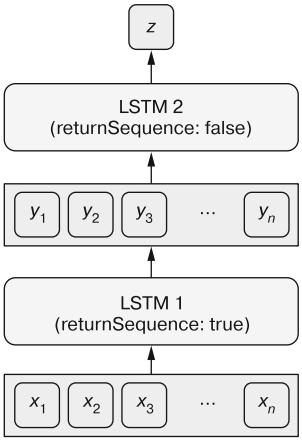


Рис. 10.2. Размещение в модели нескольких (в данном случае двух) LSTM-слоев друг над другом. Свойство returnSequence первого LSTM-слоя равно true, а потому он генерирует на выходе последовательность элементов. Последовательный выходной сигнал первого LSTM-слоя подается на вход второго. Второй LSTM-слой выдает на выходе не последовательность элементов, а один элемент, который может быть регрессионным предсказанием или массивом вероятностей из многомерной логистической функции. При этом формируется итоговый выходной сигнал модели

Листинг 10.1 содержит код, создающий модели предсказания следующего символа с архитектурой, приведенной на рис. 10.2 (отрывок из файла `lstm-text-generation/model.js`). Учтите, что, в отличие от схемы на рисунке, данный код включает плотный слой в качестве выходного слоя модели, с многомерной логистической функцией активации. Напомним, что многомерная логистическая функция активации нормализует выходные сигналы до значений от 0 до 1, в сумме равных 1, подобно распределению вероятности. Таким образом, выходной сигнал завершающего плотного слоя представляет собой предсказанные вероятности уникальных символов.

Аргумент `lstmLayerSize` функции `createModel()` служит для управления количеством LSTM-слоев и их размерами. Входную форму первого LSTM-слоя определяют настройки `sampleLen` (количество символов, получаемых моделью за один раз) и `charSetSize` (количество уникальных символов, содержащихся в текстовых данных). Для выполняемого в браузере примера `sampleLen` жестко задан равным 40; в обучающем сценарии на основе Node.js его можно задавать через флаг `--sampleLen`. Для набора данных текстов Шекспира `charSetSize` равен 71. Этот набор символов включает английские буквы в нижнем и верхнем регистрах, знаки препинания, пробел, разрыв строки и еще несколько специальных символов. При таких параметрах форма входного тензора модели, создаваемой функцией из листинга 10.1 — `[40, 71]` (не считая измерения батчей). Эта форма соответствует 40 символам в унитарном представлении. Форма выходного тензора модели — `[71]` (опять же не считая измерения батчей) — значение вероятностей многомерной логистической функции для 71 возможного варианта следующего символа.

Листинг 10.1. Создание многослойной LSTM-модели для предсказания следующего символа

```
export function createModel(sampleLen,
                            charSetSize,
                            lstmLayerSizes) {
  if (!Array.isArray(lstmLayerSizes)) {
    lstmLayerSizes = [lstmLayerSizes];
  }

  const model = tf.sequential();
  for (let i = 0; i < lstmLayerSizes.length; ++i) {
    const lstmLayerSize = lstmLayerSizes[i];
    model.add(tf.layers.lstm({
      units: lstmLayerSize,
     returnSequences: i < lstmLayerSizes.length - 1,
      inputShape: i === 0 ?
          [sampleLen, charSetSize] : undefined
    }));
  }
  model.add(
      tf.layers.dense({
        units: charSetSize,
        activation: 'softmax'
  }));

  return model;
}
```

Длина входной последовательности

Число возможных уникальных символов

Размер LSTM-слоев модели — одно число или массив чисел

Модель начинается с набора расположенных друг над другом LSTM-слоев

Задаем свойство returnSequence равным true, чтобы можно было расположить несколько LSTM-слоев друг над другом

Первый LSTM-слой отличается тем, что требует указания формы входного тензора

Модель завершается плотным слоем с многомерной логистической функцией активации по всем возможным символам, что отражает классификационную природу задачи предсказания следующего символа

Для подготовки модели к обучению скомпилируем ее, взяв в качестве функции потерь категориальную перекрестную энтропию, ведь наша модель, по существу, представляет собой классификатор с 71 классом. В качестве оптимизатора воспользуемся RMSProp, часто применяемым в рекуррентных моделях:

```
const optimizer = tf.train.rmsprop(learningRate);
model.compile({optimizer: optimizer, loss: 'categoricalCrossentropy'});
```

Данные, на которых будет обучаться модель, состоят из пар входных отрывков текста и символов, следующих за ними, закодированных в унитарные векторы (см. рис. 10.1). Описанный в файле `lstm-text-generation/data.js` класс `TextData` включает логику генерации подобных тензоров данных на основе обучающего корпуса текстов. Код здесь довольно громоздкий, но сама идея проста, она состоит в случайной выборке из очень длинной строки фрагментов фиксированной длины и преобразовании их в унитарные тензорные представления.

Если вы используете веб-версию примера, можете подобрать гиперпараметры в разделе Model Training (Обучение модели) веб-страницы, а именно: количество эпох обучения, число примеров данных из расчета на эпоху, скорость обучения и т. д. Нажмите кнопку Train Model (Обучить модель) для запуска процесса обучения модели. При обучении же с помощью Node.js эти гиперпараметры настраиваются с помощью флагов командной строки. Подробнее вы можете узнать из справочных сообщений, набрав команду `yarn train --help`.

В зависимости от заданного вами количества эпох обучения и размера модели обучение займет от нескольких минут до нескольких часов. При обучении в Node.js после каждой эпохи автоматически выводится число примеров фрагментов текста, сгенерированных моделью (табл. 10.1). По мере обучения значение потерь должно неуклонно снижаться и уменьшиться от начального значения 3,2 примерно до 1,4–1,5. Примерно после 120 эпох по мере снижения значений потерь качество сгенерированного текста улучшится до такой степени, что к концу обучения он будет выглядеть *подобным на шекспировский*, а потери на проверочном наборе данных достигнут 1,5 — не так уж далеко от неопределенности 1,3 бита/символ из эксперимента Шеннона. Но учтите, что при такой парадигме обучения и разрешающих возможностях модели сгенерированный ею текст никогда не станет похож на настоящие тексты Шекспира.

Таблица 10.1. Примеры текста, сгенерированные основанной на LSTM моделью предсказания следующего символа. Для генерации текста сначала задается начальный текст: « in hourly synod about thy particular prosperity, and lo»[1]. Для сравнения приведем фактический текст, следующий в пьесе за нашим начальным текстом: «ve thee no worse than thy old father Menenius does! ...»

Эпох обучения	Потери на проверочном наборе	T = 0	T = 0.25	T = 0.5	T = 0.75
5	2,44	"rle the the the the the the the the the the the the the the the the the the the the the the the the the the the "	"te ans and and and and and warl torle an at an yawl and tand and an an ind an an in thall ang ind an tord and and and wa"	"te toll nlatese ant ann, tomdenl, teurteeinlndting fall ald antetetell linde ing thathere taod winld mlinl theens tord y"	"p, af ane me pfleh; fove this? Iretltard efidestind ants anl het insethou loellr ard,
25	1,96	"ve tray the stanter an truent to the stanter to the stanter to the stanter to the stanter to the stanter to the stanter "	"ve to the enter an truint to the surt an truin to me truent me the will tray mane but a bean to the stanter an trust tra"	"ve of marter at it not me shank to an him truece preater the beaty atweath and that marient shall me the manst on hath s"	"rd; not an an beilloters An bentest the like have bencest on it love gray to dreath avalace the lien I am sach me, m"

Продолжение ⇩

[1] Отрывок из пьесы «Кориолан» Шекспира, акт пятый, сцена вторая. Обратите внимание, что пример текста включает разрывы строк и обрывается посередине слова (love).

Таблица 10.1 *(продолжение)*

Эпох обучения	Потери на проверочном наборе	T = 0	T = 0.25	T = 0.5	T = 0.75
50	1,67	"rds the world the world the world the world the world the world the world the world the world the worl"	"ngs they are their shall the englents the world the world the stand the provicess their string shall the world I"	"nger of the hath the forgest as you for sear the device of thee shall, them at a hame, The now the would have bo"	"ngs, he coll, As heirs to me which upon to my light fronest prowirness foir. I be chall do vall twell. SIR C"
100	1,61	"nd the sough the sought That the more the man the forth and the strange as the sought That the more the man the "	"nd the sough as the sought In the consude the more of the princes and show her art the compont "	"rds as the manner. To the charit and the stranger and house a tarron. A tommern the bear you art this a contents, "	"nd their conswents That thou be three as me a thout thou do end, The longers and an heart and not strange. A G"
120	1,49	"ve the strike the strike the strike the strike the strikes the strike And the strike the strike the strike A"	"ve the fair brother, And this in the strike my sort the strike, The strike the sound in the dear strike And "	"ve the stratter for soul. Monty to digning him your poising. This for his brother be this did fool. A mock'd"	"ve of his trusdum him. poins thinks him where sudy's such then you; And soul they will I would from in my than s"

В табл. 10.1 показаны примеры текстов при четырех различных *значениях температуры* — параметра, определяющего степень случайности сгенерированного текста. Возможно, вы заметили в примерах сгенерированного текста, что при низких значениях температуры текст выглядит более однообразным и менее естественным, в то время как бóльшим значениям соответствует менее предсказуемый текст. По умолчанию максимальное значение температуры, демонстрируемое Node.js-сценарием обучения, равно 0,75, в результате чего иногда получаются последовательности символов, напоминающие английские слова, но на самом деле ими не являющиеся (например, *stratter* и *poins* в приведенных в таблице примерах). Далее вы узнаете, как температура работает и почему так называется.

10.1.3. Температура: настройка степени стохастичности генерируемого текста

Определять выбираемый символ на каждом шаге процесса генерации текста на основе выдаваемых моделью вероятностей будет функция `sample()` (листинг 10.2). Как видите, алгоритм довольно сложен и включает вызовы трех низкоуровневых операций TensorFlow.js: `tf.div()`, `tf.log()` и `tf.multinomial()`. Почему мы используем этот сложный алгоритм, вместо того чтобы просто брать вариант с наибольшей оценкой вероятности, для чего хватило бы одного вызова `argMax()`?

Дело в том, что в данном случае выходной сигнал процесса генерации текста оказался бы *детерминированным* (deterministic). Иначе говоря, при многократных запусках результат всякий раз оказывался бы тем же самым. Все встречавшиеся нам до сих пор глубокие нейронные сети были детерминированными в том смысле, что для заданного входного тензора выходной тензор полностью определяется топологией сети и значениями весовых коэффициентов. При желании можно написать модульный тест для контроля выходного значения (см. обсуждение тестирования алгоритмов машинного обучения в главе 12). Подобный детерминизм для нашей задачи генерации текста *не* лучший вариант. В конце концов, написание текста — творческий процесс. Небольшая степень случайности сгенерированного текста при одном и том же начальном варианте делает его интереснее. Именно для этого и служат операция `tf.multinomial()` и параметр температуры. `tf.multinomial()` — источник стохастичности, степень которой определяет температура.

Листинг 10.2. Стохастическая функция выборки, с параметром температуры

Плотный слой модели выдает нормализованные оценки вероятности; чтобы преобразовать их в ненормализованные логиты, перед делением на температуру используется функция log()

```
export function sample(probs, temperature) {
  return tf.tidy(() => {
    const logPreds = tf.div(
      tf.log(probs),
      Math.max(temperature, 1e-6));
    const isNormalized = false;
    return tf.multinomial(logPreds, 1, null, isNormalized).dataSync()[0];
  });
}
```

Это небольшое положительное число служит защитой от деления на ноль. Результат деления — логиты с откорректированной степенью неопределенности

tf.multinomial() — стохастическая функция дискретизации, подобная многогранной игральной кости, вероятности выпадения разных граней которой различны и определяются logPreds — учитывающими температуру логитами

Важнейшей частью функции `sample()` в листинге 10.2 является строка:

```
const logPreds = tf.div(tf.log(probs),
                        Math.max(temperature, 1e-6));
```

Она преобразует `probs` (вероятности на выходе модели) в `logPreds` — логарифмы вероятностей, умноженные на определенный коэффициент. Что делают операции взятия логарифма (`tf.log()`) и масштабирования (`tf.div()`)? Поясним на примере.

Ради простоты допустим, что существует лишь три варианта выбора (три символа в наборе символов). Пусть наш алгоритм предсказания очередного символа для заданной входной последовательности выдал такие три оценки вероятностей:

```
[0.1, 0.7, 0.2]
```

Взглянем, как два различных значения температуры влияют на эти вероятности. Для начала возьмем относительно низкое значение: 0,25. Логиты с учетом масштабирования:

```
log([0.1, 0.7, 0.2]) / 0.25 = [-9.2103, -1.4267, -6.4378]
```

Чтобы понять, что эти логиты значат, преобразуем их обратно в оценки вероятностей с помощью уравнения для многомерной логистической функции, для чего вычислим экспоненту логитов и нормализуем их:

```
exp([-9.2103, -1.4267, -6.4378]) / sum(exp([-9.2103, -1.4267, -6.4378]))
= [0.0004, 0.9930, 0.0066]
```

Как видите, логиты для температуры 0,25 соответствуют сильно локализованному распределению вероятности, в котором вероятность второго варианта намного выше, чем у остальных двух (см. второй блок на рис. 10.3).

А что, если взять температуру повыше, скажем 0,75? Повторяя те же вычисления, получаем:

```
log([0.1, 0.7, 0.2]) / 0.75 = [-3.0701, -0.4756, -2.1459]
exp([-3.0701, -0.4756, -2.1459]) / sum([-3.0701, -0.4756, -2.1459])
= [0.0591, 0.7919 0.1490]
```

Получается распределение не с таким резким максимумом, по сравнению с предыдущим, при температуре 0,25 (см. четвертый блок на рис. 10.3). Но у него все равно более выраженный пик, чем у исходного распределения. Как вы могли догадаться, при температуре 1 получатся в точности исходные вероятности (см. рис. 10.3, пятый блок). Температура выше 1 даст более «выровненное» по вариантам распределение вероятности (см. рис. 10.3, шестой блок), хотя расстановка вариантов по местам всегда будет одинаковой.

Эти преобразованные вероятности (точнее, их логарифмы) затем передаются в функцию `tf.multinomial()`, выполняющую роль многогранной игральной кости с не равновероятными гранями, что определяется входным аргументом. В результате мы и получаем следующий символ.

Именно так параметр температуры и определяет степень стохастичности генерируемого текста. Термином *температура* мы обязаны термодинамике, из которой знаем, что чем выше температура системы, тем выше степень хаоса в ней. Такая аналогия здесь вполне уместна, ведь при повышении значения температуры в коде получается текст более хаотичного вида. Существует золотая середина значения температуры, ниже которой сгенерированный текст более однообразный и менее естественный, а выше — слишком непредсказуемый и причудливый.

На этом мы завершаем экскурс в LSTM, предназначенные для генерации текста. Обратите внимание: эта методика универсальна и применима ко многим другим типам последовательностей, с соответствующими изменениями. Например, обучив

LSTM на достаточно большом наборе музыкальных произведений, можно использовать ее для сочинения музыки, и она будет последовательно предсказывать очередную ноту по предыдущим[1].

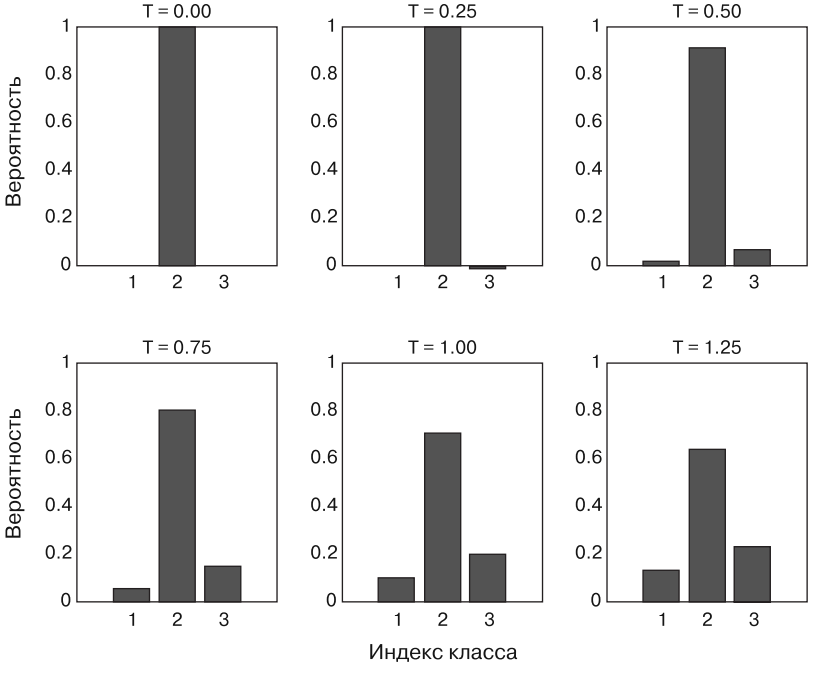


Рис. 10.3. Оценки вероятностей после масштабирования на различные значения температуры (T). Более низкое значение T ведет к более «сосредоточенному» (менее стохастическому) распределению; более высокое значение T выравнивает распределение по классам (более стохастическое распределение). Значение T, равное 1, соответствует исходным вероятностям (никаких изменений). Учтите, что относительная расстановка вариантов по местам всегда сохраняется, вне зависимости от значения T

10.2. Вариационные автокодировщики: поиск экономичного структурированного векторного представления изображений

В предыдущем разделе мы вкратце обсудили применение глубокого обучения для генерации таких последовательных данных, как текст. Теперь займемся созданием нейронных сетей для генерации изображений. Мы изучим два типа моделей: вариационный автокодировщик (variational autoencoder, VAE) и генеративную

[1] *Huang A., Wu R.* Deep Learning for Music // submitted 15 June 2016. https://arxiv.org/abs/1606.04930.

состязательную сеть (generative adversarial network, GAN). По сравнению с GAN, VAE структурно проще и насчитывает более длинную историю, а потому может послужить хорошим трамплином для прыжка в динамичный мир генерации изображений с помощью глубокого обучения.

10.2.1. Классический автокодировщик и VAE: основные понятия

На рис. 10.4 схематически показана общая архитектура автокодировщика. На первый взгляд автокодировщик кажется странной моделью, ведь размеры его входного и выходного изображений одинаковы. В простейшем случае функция потерь автокодировщика представляет собой MSE между входным и выходным сигналом. Это значит, что при обучении должным образом автокодировщик получает на входе изображение и возвращает, по существу, идентичное ему изображение. Для чего же может пригодиться такая модель?

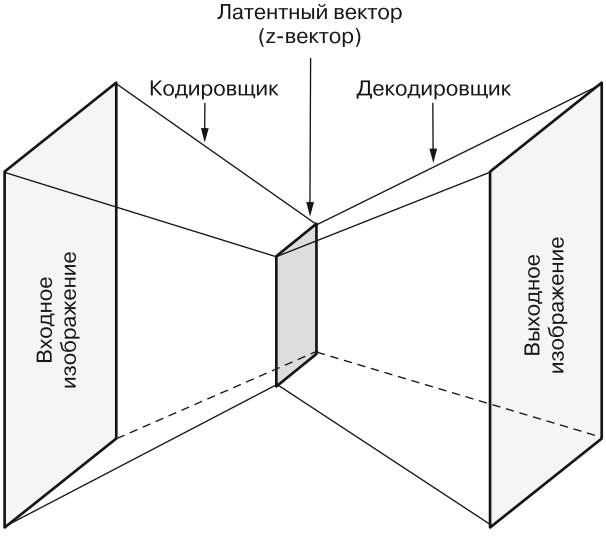


Рис. 10.4. Архитектура классического автокодировщика

На самом деле автокодировщики — важная разновидность генеративных моделей, и они отнюдь не бесполезны. Ответ на вышеупомянутый вопрос связан с их архитектурой, напоминающей по форме песочные часы (см. рис. 10.4). Самая узкая, средняя часть автокодировщика представляет собой вектор с намного меньшим, по сравнению с входным и выходным изображениями, числом элементов. А потому производимое автокодировщиком преобразование изображения в изображение нетривиально: сначала он преобразует входное изображение в очень сжатое представление, а затем восстанавливает изображение по этому представлению, без какой-либо до-

полнительной информации. Сжатое представление посередине автокодировщика называется *латентным вектором* (latent vector) или *z-вектором* (z-vector). Мы будем использовать попеременно оба термина. Векторное пространство, к которому относятся эти векторы, называется *латентным пространством* (latent space) или *z-пространством* (z-space). Часть автокодировщика, преобразующая входное изображение в латентный вектор, называется *кодировщиком*; вторая часть, преобразующая латентный вектор обратно в изображение, называется *декодировщиком*.

Латентный вектор может быть в сотни раз меньше самого изображения, как мы покажем вскоре на конкретном примере. Таким образом, кодировщик обученного автокодировщика — весьма эффективное средство понижения размерности. Создаваемая им сжатая версия входного изображения весьма лаконична, но все же содержит достаточно существенной информации для адекватного воспроизведения декодировщиком входного изображения без использования дополнительных битов информации. То, что декодировщик на это способен, также весьма примечательно.

Можно также взглянуть на автокодировщик с точки зрения теории информации. Пусть входное и выходное изображения содержат по N бит информации. При «наивном» подсчете N равно числу пикселов, умноженному на глубину цвета пикселов. Напротив, латентный вектор из-за его малого размера (скажем, m бит) из середины автокодировщика может содержать лишь малую долю этой информации. Если m меньше N, теоретически восстановить изображение из латентного вектора будет невозможно. Однако пикселы в изображении не абсолютно случайны (состоящее из совершенно случайных пикселов изображение напоминает статический шум), а следуют определенным закономерностям, например, демонстрируют однородность цветопередачи и отражают характеристики изображенных на них объектов реального мира. Так что настоящее значение N намного меньше значения, полученного в результате «наивных» вычислений, основанных на количестве и цветовой глубине пикселов. Задача автокодировщика как раз и состоит в усвоении этих закономерностей; и как раз поэтому автокодировщик способен работать.

После обучения автокодировщика можно использовать его декодировщик без кодировщика, чтобы сгенерировать по заданному латентному вектору изображения, соответствующие закономерностям и стилям обучающих изображений. Это отлично согласуется с определением генеративной модели. Более того, при некотором везении латентное пространство будет включать удобные для интерпретации структуры. В частности, все отдельные измерения латентного пространства могут соответствовать осмысленным аспектам изображения. Например, представьте себе, что мы обучили автокодировщик на изображениях человеческих лиц; тогда одно из измерений может отражать степень выраженности улыбки на лице. Если зафиксировать значения во всех прочих измерениях латентного вектора и менять только значение «измерения улыбки», декодировщик будет генерировать изображения, отличающиеся лишь широтой улыбки (см., например, рис. 10.5). Это открывает дорогу для интересных приложений, например изменения широты улыбки во входном изображении лица без изменений всего остального. Пошаговый алгоритм таков: сначала необходимо применить кодировщик и получить латентный вектор

входного изображения. Затем модифицировать только «измерение улыбки» вектора; и наконец, пропустить модифицированный латентный вектор через декодировщик.

Рис. 10.5. «Измерение улыбки». Пример усвоения автокодировщиком нужной структуры в латентном пространстве

К сожалению, *классические автокодировщики* с приведенной на рис. 10.4 архитектурой не приводят к полезным или удобно структурированным латентным пространствам. Поэтому к 2013 году они по большей части вышли из моды. VAE, изобретенные почти одновременно Дидериком Кингма и Максом Веллингом в декабре 2013 года[1] и Данило Резенде, Шакиром Мохамедом Shakir Mohamed и Дааном Вестрой в январе 2014-го[2], дополняют автокодировщики небольшой толикой статистической магии, в результате чего модели могут усваивать непрерывные и высокоструктурированные латентные пространства. VAE оказались чрезвычайно многообещающим типом генеративных моделей для изображений.

Вместо того чтобы сжимать входное изображение в фиксированный вектор в латентном пространстве, VAE превращают его в параметры статистического распределения — а именно, *гауссовского распределения*. Как вы помните из школьного курса математики, у гауссовского распределения есть два параметра — математическое ожидание и дисперсия (или, что эквивалентно, среднеквадратичное отклонение). VAE отображает каждое входное изображение в математическое ожидание этого распределения. Единственная сложность: математическое ожидание и дисперсия могут быть многомерными, если число измерений латентного пространства больше 1, как мы видели в предыдущем примере. По существу, мы полагаем, что изображения сгенерированы в ходе стохастического процесса, и хотим учитывать случайность этого процесса во время кодирования и декодирования. VAE использует параметры математического ожидания и дисперсии для случайной выборки одного вектора из распределения и декодирует его снова до размеров исходного входного сигнала (рис. 10.6). Во многом именно из-за этой стохастичности повышается устойчивость VAE к ошибкам и обеспечивается кодирование латентным пространством во всех точках осмысленных представлений: каждая выбранная точка в латентном пространстве при декодировании декодировщиком должна представлять собой допустимое выходное изображение.

---

1 *Kingma D. P., Welling M.* Auto-Encoding Variational Bayes // submitted 20 Dec. 2013. https://arxiv.org/abs/1312.6114.

2 *Rezende D. J., Mohamed S., Wierstra D.* Stochastic Backpropagation and Approximate Inference in Deep Generative Models // submitted 16 Jan. 2014. https://arxiv.org/abs/1401.4082.

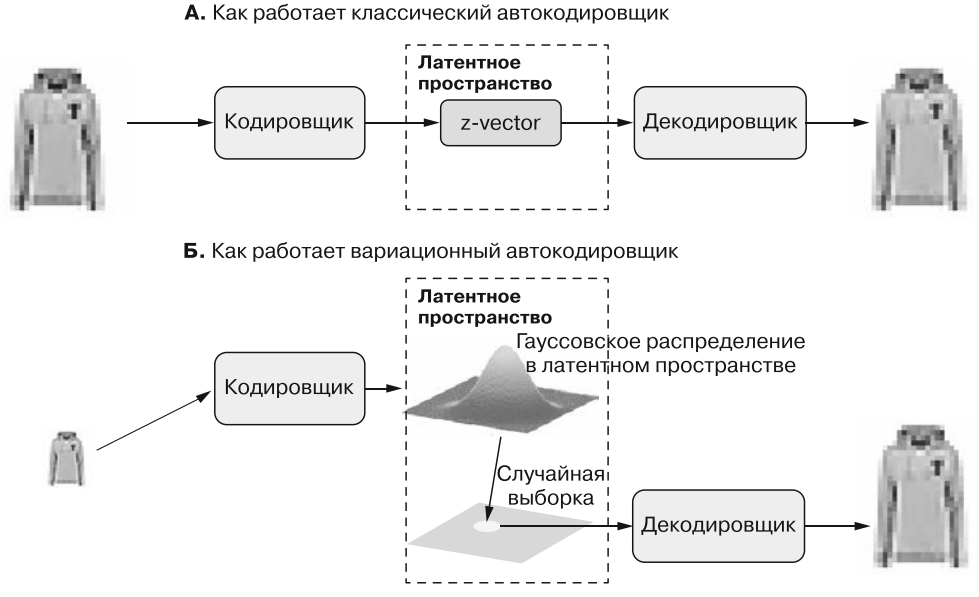


Рис. 10.6. Сравнение работы классического автокодировщика (блок А) и VAE (блок Б). Классический автокодировщик ставит в соответствие входному изображению фиксированный латентный вектор и выполняет на его основе декодирование. Напротив, VAE ставит входному изображению в соответствие распределение, описываемое математическим ожиданием и дисперсией, выбирает из этого распределения случайный латентный вектор и генерирует на его основе декодированное изображение. Изображение футболки — пример из набора данных Fashion-MNIST

Далее мы продемонстрируем вам VAE в действии на наборе данных Fashion-MNIST. Как ясно из его названия[1], прообразом для набора Fashion-MNIST[2] стал набор рукописных цифр MNIST, но он содержит изображения одежды и модных аксессуаров. Как и в наборе MNIST, данные Fashion-MNIST представляют собой изображения 28×28 в оттенках серого. Они делятся ровно на десять классов одежды и аксессуаров (например, футболки, пуловеры, туфли и сумки; см. пример на рис. 10.6). Однако набор данных Fashion-MNIST несколько сложнее для алгоритмов машинного обучения, по сравнению с набором данных MNIST. Степень безошибочности на контрольном наборе данных у самых передовых алгоритмов составляет примерно 96,5 % — намного меньше, чем 99,75%-ная безошибочность столь же продвинутых алгоритмов на наборе данных MNIST[3].

Мы создадим вариационный автокодировщик и обучим его на наборе данных Fashion-MNIST с помощью TensorFlow.js, а затем воспользуемся декодировщиком

---

1 Fashion — англ. «мода». — *Примеч. пер.*

2 *Xiao H., Rasul K., Vollgraf R.* Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms // submitted 25 Aug. 2017. https://arxiv.org/abs/1708.07747.

3 Источник: State-of-the-Art Result for All Machine Learning Problems // GitHub, 2019. http://mng.bz/6w0o.

этого VAE для выборки из двумерного латентного пространства и изучения внутренней структуры этого пространства.

10.2.2. Подробный пример VAE: пример Fashion-MNIST

Для извлечения из репозитория примера fashion-mnist-vae выполните следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/fashion-mnist-vae
yarn
yarn download-data
```

Этот пример состоит из двух частей: обучения VAE в Node.js и генерации изображений в браузере с помощью декодировщика. Для запуска обучения выполните:

```
yarn train
```

При наличии настроенного должным образом GPU с поддержкой CUDA можете воспользоваться флагом `--gpu`, чтобы повысить скорость обучения:

```
yarn train --gpu
```

На более или менее современном компьютере, оснащенном GPU с поддержкой CUDA, обучение должно занять около пяти минут и около часа — без GPU. По завершении обучения скомпонуйте и запустите браузерную клиентскую часть с помощью следующей команды:

```
yarn watch
```

Клиентская часть загрузит декодировщик VAE, сгенерирует набор изображений с помощью двумерной сетки равноудаленных латентных векторов и выведет эти изображения на странице, что даст вам понимание структуры данного латентного пространства.

Говоря техническим языком, VAE функционирует следующим образом.

1. Кодировщик преобразует входные примеры данных в два параметра в латентном пространстве: `zMean` и `zLogVar` — математическое ожидание и логарифм дисперсии соответственно. Длина обоих векторов такая же, как и размерность латентного пространства[1]. Например, при двумерном латентном пространстве `zMean` и `zLogVar` будут векторами длиной 2. Почему мы используем логарифм дисперсии, а не просто дисперсию? Потому что дисперсия по определению неотрицательна, но обеспечить выполнение подобных требований к знаку выходного сигнала не так-то просто. А логарифм дисперсии может принимать любой знак. Благодаря использованию логарифма мы можем не беспокоиться о знаке

[1] Строго говоря, ковариационная матрица латентного вектора длины N представляет собой матрицу N × N. Однако zLogVar — вектор длины N, ведь наша ковариационная матрица диагональна, то есть в нашем случае нет никакой корреляции между любыми двумя элементами латентного вектора.

выходных сигналов слоев. А логарифм дисперсии можно легко преобразовать в соответствующую дисперсию с помощью простой операции экспоненты.

2. Алгоритм VAE производит случайную выборку латентного вектора из латентного нормального распределения с помощью вектора `epsilon` — случайного вектора той же длины, что `zMean` и `zLogVar`. Этот шаг, в литературе называемый *репараметризацией* (reparametrization), можно описать простым математическим уравнением:

```
z = zMean + exp(zLogVar * 0.5) * epsilon
```

В результате умножения на 0,5 дисперсия превращается в среднеквадратичное отклонение, поскольку оно равно корню квадратному из дисперсии. Эквивалентный код на JavaScript:

```
z = zMean.add(zLogVar.mul(0.5).exp().mul(epsilon));
```

(листинг 10.3). Далее `z` подается на вход декодировщика VAE для генерации выходного изображения.

Листинг 10.3. Выборка из латентного пространства (z-пространства) с помощью пользовательского слоя

```
class ZLayer extends tf.layers.Layer {
  constructor(config) {
    super(config);
  }

  computeOutputShape(inputShape) {
    tf.util.assert(inputShape.length === 2 && Array.isArray(inputShape[0]),
        () => `Expected exactly 2 input shapes. ` +
              `But got: ${inputShape}`);
    return inputShape[0];
  }

  call(inputs, kwargs) {
    const [zMean, zLogVar] = inputs;
    const batch = zMean.shape[0];
    const dim = zMean.shape[1];
    const mean = 0;
    const std = 1.0;
    const epsilon = tf.randomNormal(
        [batch, dim], mean, std);
    return zMean.add(
        zLogVar.mul(0.5).exp().mul(epsilon));
  }

  static get ClassName() {
    return 'ZLayer';
  }
}
tf.serialization.registerClass(ZLayer);
```

Проверяем, что входной сигнал содержит ровно два значения: zMean и zLogVar

Форма выходного тензора (z) — такая же, как и форма zMean

Получаем случайный батч векторов epsilon из гауссовского распределения с нулевым математическим ожиданием и единичной дисперсией

Именно здесь происходит выборка z-векторов: zMean + standardDeviation * epsilon

На случай необходимости сериализации слоя задаем статическое свойство ClassName

Регистрируем класс для обеспечения возможности десериализации

В нашей реализации VAE шаг дискретизации латентного вектора выполняется пользовательским слоем `ZLayer` (см. листинг 10.3). Мы уже встречали пользовательские слои TensorFlow.js в главе 9 (слой `GetLastTimestepLayer`, использовавшийся в алгоритме преобразования дат на основе механизма внимания). Пользовательский слой нашего VAE несколько сложнее и заслуживает небольших пояснений.

Класс `ZLayer` содержит два ключевых метода: `computeOutputShape()` и `call()`. `computeOutputShape()` используется TensorFlow.js для определения выходной формы экземпляра `Layer` по форме (-ам) входного сигнала. А сама математическая логика заключена в методе `call()`. Он содержит вышеприведенную строку с уравнением. Следующий код взят из файла `fashion-mnist-vae/model.js`.

Как демонстрирует листинг 10.4, сначала создается экземпляр `ZLayer` для использования в качестве части кодировщика. Кодировщик написан в виде функциональной модели, а не более простой последовательной модели, из-за нелинейной внутренней структуры и трех выходных сигналов: `zMean`, `zLogVar` и `z` (рис. 10.7). Кодировщик выдает на выходе `z` для дальнейшего использования декодировщиком, но зачем выходной сигнал кодировщика включает `zMean` и `zLogVar`? Дело в том, что они в дальнейшем применяются для вычисления функции потерь VAE, как вы скоро увидите.

Листинг 10.4. Кодировщик нашего VAE (выдержка из файла fashion-mnist-vae/model.js)

```
function encoder(opts) {
  const {originalDim, intermediateDim, latentDim} = opts;

  const inputs = tf.input({shape: [originalDim], name: 'encoder_input'});
  const x = tf.layers.dense({units: intermediateDim, activation: 'relu'})
                .apply(inputs);
  const zMean = tf.layers.dense({units: latentDim, name: 'z_mean'}).apply(x);
  const zLogVar = tf.layers.dense({
        units: latentDim,
        name: 'z_log_var'
      }).apply(x);
  const z = new ZLayer({name: 'z',
        outputShape: [latentDim]}).apply([zMean, zLogVar]);

  const enc = tf.model({
    inputs: inputs,
    outputs: [zMean, zLogVar, z],
    name: 'encoder',
  })
  return enc;
}
```

В основе кодировщика лежит простой MLP с одним скрытым слоем

В отличие от обычного MLP он содержит после скрытого плотного слоя два слоя для предсказания zMean и zLogVar соответственно. Кроме того, именно поэтому мы используем функциональную модель вместо более простой последовательной модели

Создает экземпляр нашего пользовательского Zlayer и производит с его помощью выборку случайных примеров данных, соответствующих задаваемому zMean и zLogVar распределению

Помимо `ZLayer`, кодировщик включает два многослойных перцептрона, по одному скрытому слою каждый, используемых для преобразования схлопнутых входных изображений Fashion-MNIST в векторы `zMean` и `zLogVar` соответственно. Скрытый слой у этих MLP один, но выходные слои — отдельные. Подобная ветвящаяся топология модели возможна благодаря тому, что кодировщик — функциональная модель.

Рис. 10.7. Схематическая иллюстрация реализации VAE в TensorFlow.js, включая внутреннее устройство кодировщика и декодировщика, а также пользовательской функции потерь и оптимизатора, участвующих в обучении VAE

Код в листинге 10.5 создает декодировщик. Его топология проще, чем у кодировщика: для преобразования входного z-вектора в изображение той же формы, что и входной сигнал кодировщика, используется MLP. Учтите, что наш VAE обрабатывает изображения несколько упрощенным и нестандартным образом, схлопывая изображения в одномерные векторы, а потому отбрасывая пространственную информацию. В ориентированных на обработку изображений VAE обычно используются сверточные слои и слои субдискретизации, но в силу простоты наших изображений (их маленького размера и наличия всего одного цветового канала) для задач нашего примера вполне достаточно подхода со схлопыванием.

Листинг 10.5. Декодировщик нашего VAE (выдержка из файла fashion-mnist-vae/model.js)

```
function decoder(opts) {
  const {originalDim, intermediateDim, latentDim} = opts;

  const dec = tf.sequential({name: 'decoder'});
  dec.add(tf.layers.dense({
    units: intermediateDim,
    activation: 'relu',
    inputShape: [latentDim]
  }));
  dec.add(tf.layers.dense({
    units: originalDim,
    activation: 'sigmoid'
  }));
  return dec;
}
```

Декодировщик представляет собой простой MLP, преобразующий латентный (z-) вектор в (схлопнутое) изображение

Сигма-функция активации отлично подходит для выходного слоя, гарантируя, что значения символов ограничиваются интервалом от 0 до 1

Для объединения кодировщика и декодировщика в единый объект `tf.LayerModel` — VAE код из листинга 10.6 извлекает третий выходной сигнал (z-вектор) кодировщика и пропускает его через декодировщик. Затем объединенная модель выдает на выходе декодированное изображение вместе с еще тремя выходными сигналами: `zMean`, `zLogVar` и z-векторами. На этом завершается описание топологии модели VAE. Для обучения модели нам нужны еще функция потерь и оптимизатор. Код в листинге 10.6 взят из файла `fashion-mnist-vae/model.js`.

Листинг 10.6. Объединяем кодировщик и декодировщик в VAE

```
function vae(encoder, decoder) {
  const inputs = encoder.inputs;
  const encoderOutputs = encoder.apply(inputs);
  const encoded = encoderOutputs[2];
  const decoderOutput = decoder.apply(encoded);
  const v = tf.model({
    inputs: inputs,
    outputs: [decoderOutput, ...encoderOutputs],
    name: 'vae_mlp',
  })
  return v;
}
```

Входной сигнал VAE — тот же, что и входной сигнал кодировщика: исходное входное изображение

Только последний (z) из входных сигналов кодировщика попадает в декодировщик

Выходной сигнал объекта модели VAE включает декодированное изображение, помимо zMean, zLogVar и z

Из-за нелинейной топологии модели мы используем функциональный API

При обсуждении модели simple-object-detection в главе 5 мы рассказывали, как в TensorFlow.js можно описать пользовательскую функцию потерь. Здесь для обучения VAE нам как раз нужна пользовательская функция потерь, поскольку она будет суммой двух составляющих: одна выражает количественно расхождение входного и выходного сигналов, а вторая — статистические свойства латентного пространства. Это напоминает пользовательскую функцию потерь модели simple-object-detection, которая была суммой составляющей, отвечающей за классификацию объекта, и второй, отвечающей за определение его местоположения.

Как вы видите из кода в листинге 10.7 (из файла `fashion-mnist-vae/model.js`), описание члена расхождения входного и выходного сигналов не доставляет сложностей: мы просто вычисляем MSE между исходным входным и выходным сигналами декодировщика. Однако статистическая составляющая — так называемое *расстояние Кульбака — Лейблера* (Kullbach-Leibler divergence) — сложнее с математической точки зрения. Мы избавим вас от математических подробностей[1], но на интуитивном уровне компонент расстояния Кульбака — Лейблера (`klLoss` в приведенном коде) обеспечивает более равномерное распределение различных входных изображений вокруг центра латентного пространства, что упрощает для декодировщика задачу их интерполяции. Таким образом, компонент `klLoss` можно считать своего рода членом уравнения, отвечающим за регуляризацию, дополнительно к основному члену, отвечающему в VAE за расхождение входного и выходного сигналов.

Листинг 10.7. Функция потерь для VAE

```
function vaeLoss(inputs, outputs) {
  const originalDim = inputs.shape[1];
  const decoderOutput = outputs[0];
  const zMean = outputs[1];
  const zLogVar = outputs[2];

  const reconstructionLoss =
      tf.losses.meanSquaredError(inputs, decoderOutput).mul(originalDim);

  let klLoss = zLogVar.add(1).sub(zMean.square()).sub(zLogVar.exp());
  klLoss = klLoss.sum(-1).mul(-0.5);
  return reconstructionLoss.add(klLoss).mean();
}
```

Вычисляет член уравнения, отвечающий за восстановление изображения. Цель минимизации этого члена — добиться соответствия выходных сигналов модели входным данным

Суммируя потери — член восстановления изображения и член расстояния Кульбака — Лейблера — получаем итоговые потери VAE

Вычисляет расстояние Кульбака — Лейблера между zLogVar и zMean. Цель минимизации этого члена — обеспечить более близкое к нормальному распределение латентной величины вокруг центра латентного пространства

Еще одна недостающая составляющая, необходимая для обучения нашего VAE, — оптимизатор и использующий его шаг обучения. Мы воспользуемся популярным оптимизатором ADAM (`tf.train.adam()`). Шаг обучения VAE отличается от обучения всех прочих моделей в этой книге тем, что не использует методов `fit()` или `fitDataset()` объекта модели, а вызывает метод `minimize()` оптимизатора

[1] Более детальное описание расстояния Кульбаха — Лейблера вы можете найти в следующем сообщении из блога Ирхума Шавката: http://mng.bz/vlvr.

(листинг 10.8). Дело в том, что член расстояния Кульбака — Лейблера использует два из четырех выходных сигналов модели, а в TensorFlow.js методы `fit()` и `fitDataset()` работают лишь в том случае, если функция потерь каждого из выходных сигналов модели не зависит ни от какого другого выходного сигнала.

Как видно из листинга 10.8, функция `minimize()` вызывается с единственным аргументом — стрелочной функцией. Эта стрелочная функция возвращает потери для текущего батча схлопнутых изображений (`reshaped` в коде), для которого является замыканием. Функция `minimize()` вычисляет градиенты функции потерь относительно всех обучаемых весовых коэффициентов VAE (включая кодировщик и декодировщик), подстраивая их согласно алгоритму ADAM, после чего применяет обновления к весам в направлениях, противоположных этим градиентам. На этом отдельный шаг обучения завершается. Отдельная эпоха обучения состоит из многократного выполнения этого шага по всем изображениям из набора данных Fashion-MNIST. Команда `yarn train` выполняет несколько эпох обучения (по умолчанию 5), после чего значение функции потерь сходится и декодировщик VAE сохраняется на диск. Кодировщик не сохраняется потому, что не используется в следующем шаге, в браузере.

Листинг 10.8. Цикл обучения VAE (фрагмент из файла fashion-mnist-vae/train.js)

```
for (let i = 0; i < epochs; i++) {
  console.log(`\nEpoch #${i} of ${epochs}\n`)
  for (let j = 0; j < batches.length; j++) {
    const currentBatchSize = batches[j].length
    const batchedImages = batchImages(batches[j]);

    const reshaped =
        batchedImages.reshape([currentBatchSize, vaeOpts.originalDim]);

    optimizer.minimize(() => {
      const outputs = vaeModel.apply(reshaped);
      const loss = vaeLoss(reshaped, outputs, vaeOpts);
      process.stdout.write('.');
      if (j % 50 === 0) {
        console.log('\nLoss:', loss.dataSync()[0]);
      }
      return loss;
    });
    tf.dispose([batchedImages, reshaped]);
  }
  console.log('');
  await generate(decoderModel, vaeOpts.latentDim);
}
```

Получаем батч (схлопнутых) изображений набора Fashion-MNIST

Отдельный шаг обучения VAE: выполнение предсказания с помощью VAE и вычисление функции потерь для подстройки с помощью optimizer.minimize всех обучаемых весовых коэффициентов модели

В конце каждой эпохи обучения генерирует изображение с помощью декодировщика и выводит его в консоль для предварительного просмотра

Поскольку мы не используем стандартный метод fit(), то не можем и воспользоваться встроенным индикатором хода выполнения, а потому нам приходится самим выводить обновления состояния в консоль

Открываемая командой `yarn watch` веб-страница загружает сохраненный декодировщик и генерирует с его помощью таблицу изображений, примерно таких,

как показано на рис. 10.8. Эти изображения получены из равномерной координатной сетки латентных векторов двумерного латентного пространства. В UI есть возможность задания верхней и нижней границ по каждому из двух латентных измерений.

Полученная таблица изображений демонстрирует вполне непрерывное распределение различных типов одежды из набора данных Fashion-MNIST, где один тип одежды плавно переходит в другой по мере движения по непрерывному пути в латентном пространстве (например, пуловер в футболку, футболка в брюки, ботинки в туфли). Обратите внимание на осмысленность отдельных направлений латентного пространства внутри его подобластей. Например, в районе верхней части латентного пространства горизонтальное измерение, по-видимому, отражает «ботинкообразность или туфлеобразность»; в нижнем правом углу латентного пространства горизонтальное измерение, по-видимому, отражает «футболкообразность» или «брюкообразность» и т. д.

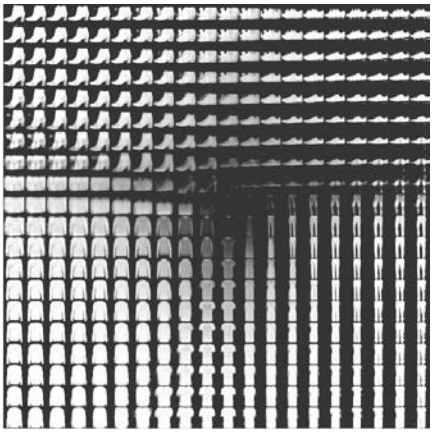


Рис. 10.8. Выборка из латентного пространства VAE после обучения. На этом рисунке приведена таблица 20 × 20 выходных сигналов декодировщика, соответствующая равномерной координатной сетке 20 × 20 двумерных латентных векторов, значения измерений которых входят в интервал [–4, 4]

В следующем разделе мы рассмотрим другой важнейший тип моделей для генерации изображений: GAN.

10.3. Генерация изображений с помощью GAN

С тех пор как Иэн Гудфеллоу (Ian Goodfellow) и его соратники представили GAN в 2014[1], интерес к этой методике начал быстро расти, а она сама — развиваться. Сегодня генеративные состязательные сети — мощное средство генерации изображений и прочих типов данных, способное генерировать изображения высокой четкости, порой неотличимые человеческим глазом от настоящих. Взгляните, например, на изображения человеческих лиц, сгенерированные StyleGAN компании NVIDIA, на рис. 10.9[2]. Если бы не редкие артефакты на лицах и ненатурально выглядящий фон, человеку практически невозможно было бы отличить эти сгенерированные изображения от настоящих.

---

[1] *Goodfellow I. et al.* Generative Adversarial Nets // NIPS Proceedings, 2014. http://mng.bz/4ePv.

[2] Сайт по адресу https://thispersondoesnotexist.com. Научная статья: *Karras T., Laine S., Aila T.* A Style-Based Generator Architecture for Generative Adversarial Networks // submitted 12 Dec. 2018. https://arxiv.org/abs/1812.04948.

Рис. 10.9. Избранные примеры изображений человеческих лиц с сайта https://thispersondoesnotexist.com (апрель 2019 года); сгенерированы StyleGAN компании NVIDIA

Помимо генерации убедительно выглядящих изображений «с потолка», можно генерировать с помощью GAN изображения, соответствующие определенным входным данным или параметрам, что ведет к множеству интересных приложений для конкретных задач. Например, GAN можно использовать для генерации изображений в высоком разрешении на основе входных сигналов низкого разрешения (технология повышения разрешения (image super-resolution)), заполнения отсутствующих частей изображений (ретуширование изображений (image inpainting)), преобразования черно-белого изображения в цветное (колоризация изображений (image colorization)), генерации изображений по текстовому описанию и генерации человека в заданной позе по входному изображению того же человека в другой позе. Кроме того, были разработаны новые типы GAN для генерации других выходных сигналов, кроме изображений, например музыки[1]. Помимо очевидной пользы генерации неограниченных объемов правдоподобных изображений для различных областей искусства и дизайна компьютерных игр, у GAN есть и другие приложения, например генерация обучающих примеров данных для глубокого обучения в случае, когда получение настоящих примеров связано с большими затратами. Например, GAN используются при генерации правдоподобных уличных пейзажей для обучения нейронных сетей беспилотных автомобилей[2].

Хотя VAE и GAN — генеративные модели, в их основе лежат различные идеи. В то время как вариационные автокодировщики обеспечивают качество сгенерированных изображений, используя среднеквадратичное расхождение исходного входного сигнала и выходного сигнала декодировщика в качестве функции потерь, GAN применяет *дискриминатор* (discriminator), как мы скоро расскажем. Кроме того, входными данными во многих вариантах GAN могут быть не только векторы из латентных пространств, но и условные входные сигналы, например желаемый класс изображения. Хороший пример этого — ACGAN, который мы обсудим далее.

---

[1] См. проект MuseGAN, созданный Хао-Вен Дуном и др.: https://salu133445.github.io/musegan/.

[2] *Vincent J.* Nvidia Uses AI to Make it Snow on Streets that Are Always Sunny // The Verge, 5 Dec. 2017. http://mng.bz/Q0oQ.

В этом типе GAN со смешанными входными сигналами латентные пространства уже не непрерывны относительно входных сигналов сети.

В данном разделе мы подробнее обсудим относительно простой тип GAN. А именно, поговорим о применении *вспомогательного классификатора* GAN (auxiliary classifier GAN, ACGAN)[1] для уже знакомого нам набора данных рукописных цифр MNIST. Благодаря этому мы получим модель, способную генерировать изображения цифр, в точности напоминающие настоящие цифры набора MNIST. В то же время мы сможем сами контролировать, к какому классу цифр (от 0 до 9) принадлежат сгенерированные изображения, благодаря вспомогательному классификатору ACGAN. Чтобы разобраться, как функционирует ACGAN, пройдемся по этому алгоритму пошагово. Во-первых, мы расскажем, как работает основная часть ACGAN — GAN. А затем опишем дополнительные механизмы, позволяющие задавать классы цифр.

10.3.1. Основная идея GAN

Как GAN обучается генерировать правдоподобные изображения? Благодаря взаимодействию двух его составляющих: *генератора* (generator) и *дискриминатора* (discriminator). Можно провести аналогию генератора с фальсификатором, цель которого — нарисовать высококачественные имитации картин Пикассо, а дискриминатора — с торговцем произведениями искусства, задача которого — отличить поддельные картины Пикассо от подлинников. Фальсификатор (генератор) пытается создать все более и более правдоподобные подделки, чтобы обмануть торговца картинами (дискриминатор), а торговец картинами стремится все лучше и лучше анализировать картины, чтобы *не* дать фальсификатору себя обмануть. Это соперничество двух игроков и послужило источником эпитета «состязательная» в названии GAN. Что любопытно, фальсификатор и торговец картинами в конечном итоге *помогают* друг другу совершенствовать навыки, несмотря на явное соперничество.

Изначально фальсификатор плохо подделывает картины Пикассо — из-за случайных начальных значений весовых коэффициентов. В результате торговец картинами (дискриминатор) быстро обучается различать настоящие и поддельные картины. Важный нюанс работы алгоритма: каждый раз, принося новую картину торговцу, фальсификатор получает подробную обратную связь (от торговца) о том, какие фрагменты картины выглядят неправильно и как поменять картину, чтобы она стала более правдоподобной. Фальсификатор усваивает и запоминает эту информацию, так что в следующий раз приносит торговцу чуть лучше выглядящие картины. Этот процесс повторяется многократно. Оказывается, если задать все параметры должным образом, в итоге наш фальсификатор (генератор) станет настоящим мастером своего дела. Конечно, в результате дискриминатор (торговец картинами) также станет мастером своего дела, но обычно после обучения GAN нужен только генератор.

На рис. 10.10 приведена более подробная иллюстрация обучения дискриминатора универсальной модели GAN. Для обучения дискриминатора необходим

[1] *Odena A., Olah C., Shlens J.* Conditional Image Synthesis with Auxiliary Classifier GANs // submitted 30 Oct. 2016, https://arxiv.org/abs/1610.09585.

батч сгенерированных (генератором) изображений и батч настоящих изображений. Но генератор не может брать изображения «с потолка», в качестве входного сигнала ему необходим какой-то вектор. Латентные векторы, по сути, аналогичны использовавшимся для VAE в разделе 10.2. Латентный вектор для каждого сгенерированного генератором изображения представляет собой одномерный тензор формы `[latentSize]`. Но, как и в большинстве процедур обучения в книге, данный шаг выполняется над батчем изображений за раз. Поэтому форма латентного вектора: `[batchSize, latentSize]`. А настоящие изображения мы берем непосредственно из самого набора данных MNIST. Ради симметрии на каждом шаге обучения мы выбираем `batchSize` настоящих изображений (ровно столько же, сколько и сгенерированных).

Сгенерированные и настоящие изображения объединяются в один батч изображений, представленный в виде тензора формы `[2 * batchSize, 28, 28, 1]`. К этому объединенному батчу изображений применяется дискриминатор, выдающий на выходе предсказанные оценки вероятности реальности отдельных изображений, которые легко можно сверить с контрольными значениями (мы знаем, какие изображения настоящие, а какие — сгенерированные!) с помощью функции потерь на основе бинарной перекрестной энтропии. Далее в дело вступает уже привычный нам алгоритм обратного распространения ошибки, который обновляет весовые параметры дискриминатора с помощью оптимизатора (не показанного на рисунке), немного подталкивая дискриминатор в сторону правильных предсказаний. Учтите, что единственный вклад генератора в этот шаг обучения состоит в предоставлении сгенерированных примеров данных, но в процессе обратного распространения ошибки он не обновляется. Обновление генератора происходит на следующем шаге обучения (рис. 10.11).

Рисунок 10.11 иллюстрирует шаг обучения генератора. Генератор получает возможность сделать еще один батч сгенерированных изображений. Но в отличие от обучения дискриминатора никакие настоящие изображения MNIST нам не нужны. Дискриминатор получает на входе этот батч сгенерированных изображений вместе с батчем бинарных меток реальности изображений. Задавая значения всех меток реальности равными 1, мы *притворяемся*, что сгенерированные изображения настоящие. Конечно, изображения сгенерированные (ненастоящие), но пусть метки реальности все равно указывают, что они настоящие. Дискриминатор может (совершенно правильно) присвоить низкие значения вероятностей реальности некоторым из входных изображений, но при этом значение функции потерь на основе бинарной перекрестной энтропии будет большим благодаря фальшивым меткам реальности, и в результате обратного распространения ошибки генератор будет обновлен так, что метки реальности дискриминатора чуть-чуть увеличатся. Обратите внимание, что процесс обратного распространения ошибки обновляет *только* генератор, оставляя дискриминатор без изменений. Это еще одна важная хитрость, благодаря которой не дискриминатор снижает планку реалистичности изображений, а генератор создает чуть более реалистичные изображения. Такой эффект достигается за счет блокирования дискриминатора модели — эта операция применялась при переносе обучения в главе 5.

Рис. 10.10. Схематическая иллюстрация алгоритма обучения дискриминатора GAN. Учтите, что на этой схеме ради простоты опущена часть ACGAN, относящаяся к классам цифр. Полную схему обучения дискриминатора ACGAN вы можете найти на рис. 10.13

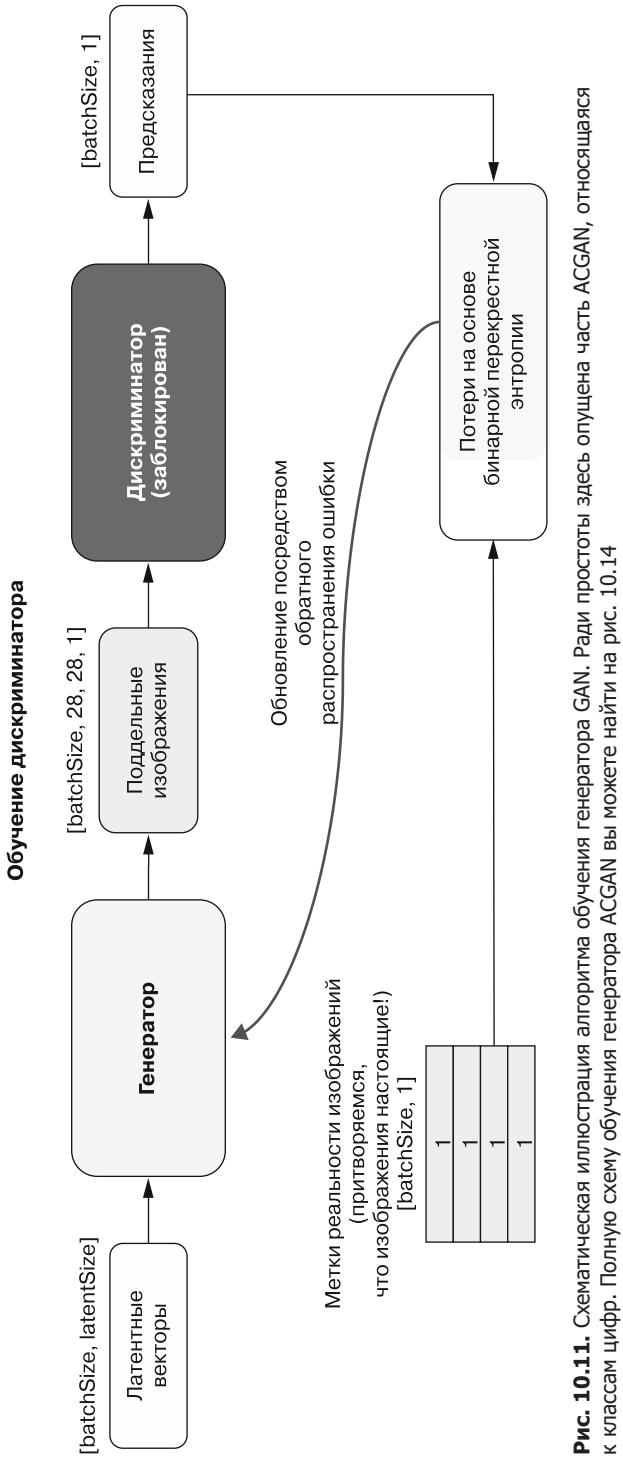


Рис. 10.11. Схематическая иллюстрация алгоритма обучения генератора GAN. Ради простоты здесь опущена часть ACGAN, относящаяся к классам цифр. Полную схему обучения генератора ACGAN вы можете найти на рис. 10.14

Подытожим шаг обучения генератора: мы блокируем дискриминатор и подаем метки реальности из сплошных единиц для вычисления функции потерь, несмотря на то что его входные изображения фальшивые, сгенерированные генератором. В результате весовые коэффициенты генератора обновляются так, что он начинает генерировать изображения, выглядящие для дискриминатора чуть более правдоподобно. Обучить генератор подобным образом можно, только если дискриминатор достаточно хорошо отличает, какие изображения настоящие, а какие — сгенерированные. Как добиться этого? С помощью шага обучения дискриминатора, который мы упоминали выше. Таким образом, эти два шага обучения переплетаются, как инь и ян, и эти две части GAN борются и помогают друг другу одновременно.

На этом наш рассказ в общих чертах про обучение GAN завершается. В следующем разделе мы рассмотрим внутренние структуры дискриминатора и генератора и посмотрим на включение в них информации о классе изображения.

10.3.2. «Кирпичики» GAN

В листинге 10.9 приведен код TensorFlow.js для создания дискриминатора ACGAN MNIST (из файла `mnist-acgan/gan.js`). В основе дискриминатора — глубокая нейронная сеть, подобная тем, с которыми мы встречались в главе 4. Форма ее входного сигнала традиционная для изображений MNIST, а именно `[28, 28, 1]`. Входное изображение проходит через четыре слоя двумерной свертки (conv2d), а затем схлопывается и подвергается обработке в двух плотных слоях. Один плотный слой выдает бинарное предсказание реальности входного изображения, а второй — вероятности десяти выходных классов из многомерной логистической функции. Дискриминатор представляет собой функциональную модель с выходными сигналами обоих плотных слоев. Топология дискриминатора с одним входным и двумя выходными сигналами схематично представлена в блоке A на рис. 10.12.

Листинг 10.9. Создание дискриминатора ACGAN

```
function buildDiscriminator() {
  const cnn = tf.sequential();

  cnn.add(tf.layers.conv2d({
    filters: 32,
    kernelSize: 3,
    padding: 'same',
    strides: 2,
    inputShape: [IMAGE_SIZE, IMAGE_SIZE, 1]
  }));

  cnn.add(tf.layers.leakyReLU({alpha: 0.2}));
  cnn.add(tf.layers.dropout({rate: 0.3}));

  cnn.add(tf.layers.conv2d(
      {filters: 64, kernelSize: 3, padding: 'same', strides: 1}));
  cnn.add(tf.layers.leakyReLU({alpha: 0.2}));
  cnn.add(tf.layers.dropout({rate: 0.3}));
```

Дискриминатор получает на входе только один сигнал: изображения в формате MNIST

Слои дропаута служат для борьбы с переобучением

```
  cnn.add(tf.layers.conv2d(
      {filters: 128, kernelSize: 3, padding: 'same', strides: 2}));
  cnn.add(tf.layers.leakyReLU({alpha: 0.2}));
  cnn.add(tf.layers.dropout({rate: 0.3}));

  cnn.add(tf.layers.conv2d(
      {filters: 256, kernelSize: 3, padding: 'same', strides: 1}));
  cnn.add(tf.layers.leakyReLU({alpha: 0.2}));
  cnn.add(tf.layers.dropout({rate: 0.3}));

  cnn.add(tf.layers.flatten());

  const image = tf.input({shape: [IMAGE_SIZE, IMAGE_SIZE, 1]});
  const features = cnn.apply(image);

  const realnessScore =
      tf.layers.dense({units: 1, activation: 'sigmoid'}).apply(features);
  const aux = tf.layers.dense({units: NUM_CLASSES, activation: 'softmax'})
                  .apply(features);

  return tf.model({inputs: image, outputs: [realnessScore, aux]});
}
```

Первый из двух выходных сигналов дискриминатора: оценка вероятности, полученная в результате бинарной классификации подлинности изображения

Второй выходной сигнал представляет собой вероятности для десяти классов цифр MNIST, полученные на основе многомерной логистической функции

Код в листинге 10.10 отвечает за создание генератора ACGAN. Как мы упоминали ранее, генератор для генерации изображений требует *латентного вектора* (в коде — `latent`) в качестве входного сигнала, что отражено в параметре `inputShape` его первого плотного слоя. Впрочем, если вы внимательно взглянете на код, то увидите, что генератор на самом деле получает *два* входных сигнала, что показано в блоке Б на рис. 10.12. Помимо латентного вектора — тензора формы `[latentSize]`, генератору необходим еще один входной сигнал — `imageClass` — формы `[1]`. С его помощью мы сообщаем модели, какой класс цифры MNIST (от 0 до 9) она должна сгенерировать. Например, если нам нужно, чтобы модель сгенерировала изображение для цифры 8, необходимо передать в качестве второго входного сигнала тензорное значение `tf.tensor2d([[8]])` (учтите, что модель всегда ожидает на входе батчи, даже если передается только один пример данных). Аналогично если модель должна сгенерировать два изображения: одно для цифры 8, а второе для 9, необходимо передать тензор `tensor2d([[8], [9]])`. При получении генератором входного сигнала `imageClass` слой вложений преобразует его в тензор той же формы, что и `latent` (`[latentSize]`). Этот шаг математически подобен процедуре поиска вложений из моделей анализа тональностей и преобразования дат (см. главу 9). Требуемый класс цифры выражается количественно аналогично индексам слов в примере анализа тональностей и индексам символов в примере преобразования дат. Он преобразуется в одномерный вектор так же, как в одномерные векторы преобразовывались индексы слов и символов. Однако цель поиска вложений для `imageClass` здесь иная: для объединения его с вектором `latent` в общий вектор (который в листинге 10.10 называется `h`) с помощью слоя `multiply`, выполняющего поэлементное умножение двух векторов одинаковой формы. Полученный тензор такой же формы, как и у входных сигналов (`[latentSize]`), поступает на вход дальнейших частей генератора.

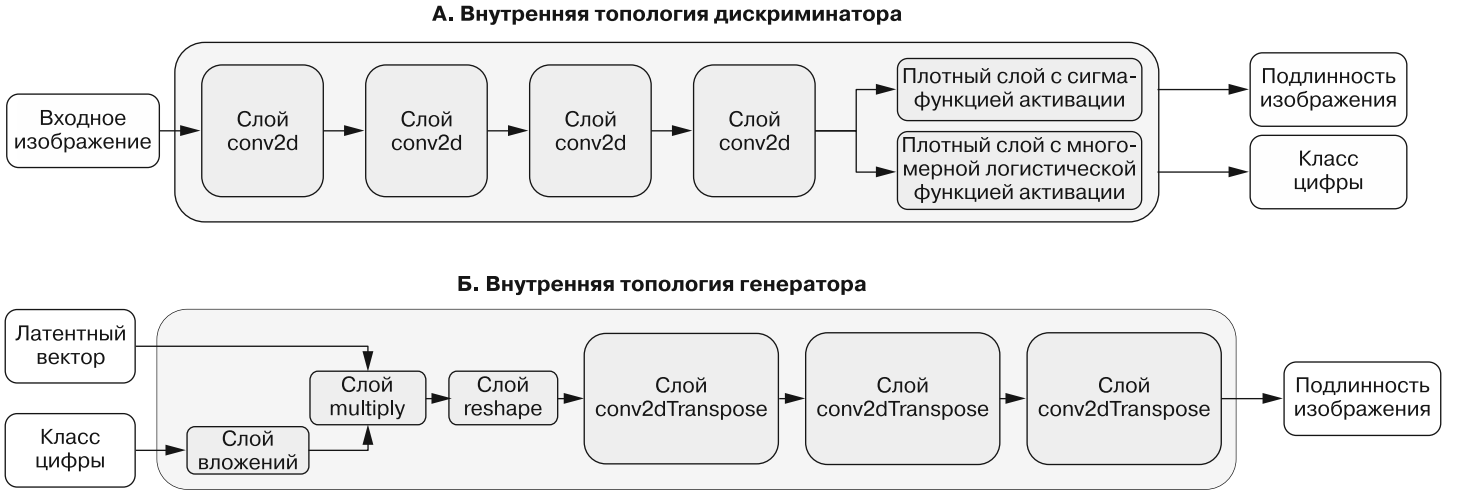


Рис. 10.12. Схематическая иллюстрация внутренней топологии дискриминатора (блок А) и генератора (блок Б) ACGAN. Для простоты некоторые нюансы (слои дропаута в дискриминаторе) опущены. См. подробный код в листингах 10.9 и 10.10

Генератор сразу же применяет к объединенному латентному вектору `h` плотный слой и приводит его к трехмерной форме `[3, 3, 384]`. В результате этого изменения формы получается напоминающий изображение тензор, который дальнейшие части генератора смогут преобразовать в изображение классической для MNIST формы (`[28, 28, 1]`).

Вместо привычных слоев conv2d для преобразования входного сигнала генератор использует слой conv2dTranspose для преобразования тензоров изображений. В общих чертах, слой conv2dTranspose выполняет обратную слою conv2d операцию (иногда называемую *обратной сверткой* (deconvolution)). Высота и ширина выходного сигнала слоя conv2d обычно меньше, чем входного (за исключением редких случаев, когда `kernelSize = 1`), как вы видели в сверточных сетях в главе 4. Высота и ширина же выходного сигнала слоя conv2dTranspose обычно больше, чем входного. Другими словами, если слой conv2d обычно *уплотняет* измерения входного сигнала, то слой conv2dTranspose чаще всего *расширяет* их. Именно поэтому в генераторе первый слой conv2dTranspose получает входной сигнал высотой 3 и шириной 3, а последний слой conv2dTranspose выдает на выходе сигнал высотой 28 и шириной 28. Таким образом генератор преобразует входной латентный вектор и индекс цифры в изображение со стандартными для MNIST размерами. Код в следующем листинге взят из файла `mnist-acgan/gan.js`; ради большей ясности кода мы убрали часть кода проверки ошибок.

Листинг 10.10. Создание генератора ACGAN

```
function buildGenerator(latentSize) {
  const cnn = tf.sequential();

  cnn.add(tf.layers.dense({
    units: 3 * 3 * 384,
    inputShape: [latentSize],
    activation: 'relu'
  }));

  cnn.add(tf.layers.reshape({targetShape: [3, 3, 384]}));

  cnn.add(tf.layers.conv2dTranspose({
    filters: 192,
    kernelSize: 5,
    strides: 1,
    padding: 'valid',
    activation: 'relu',
    kernelInitializer: 'glorotNormal'
  }));
  cnn.add(tf.layers.batchNormalization());

  cnn.add(tf.layers.conv2dTranspose({
    filters: 96,
    kernelSize: 5,
    strides: 2,
    padding: 'same',
```

Количество нейронов выбирается так, чтобы форма полученного в результате изменения формы выходного сигнала и применения к нему последующих слоев conv2dTranspose тензора в точности соответствовала форме изображений MNIST [28, 28, 1]

Повышающая дискретизация, с [3, 3...] до [7, 7...]

Повышающая дискретизация до [14, 14...]

```
    activation: 'relu',
    kernelInitializer: 'glorotNormal'
  }));
  cnn.add(tf.layers.batchNormalization());

  cnn.add(tf.layers.conv2dTranspose({
    filters: 1,
    kernelSize: 5,
    strides: 2,
    padding: 'same',
    activation: 'tanh',
    kernelInitializer: 'glorotNormal'
  }));

  const latent = tf.input({shape: [latentSize]});

  const imageClass = tf.input({shape: [1]});

  const classEmbedding = tf.layers.embedding({
    inputDim: NUM_CLASSES,
    outputDim: latentSize,
    embeddingsInitializer: 'glorotNormal'
  }).apply(imageClass);

  const h = tf.layers.multiply().apply(

      [latent, classEmbedding]);

  const fakeImage = cnn.apply(h);
  return tf.model({
   inputs: [latent, imageClass],
   outputs: fakeImage
  });
}
```

Повышающая дискретизация до [28, 28...]

Первый из двух входных сигналов генератора: латентный (из z-пространства) вектор, используемый в качестве начального значения для генерации поддельных изображений

Второй входной сигнал генератора: метки классов, определяющие, к какому из десяти классов цифр MNIST должны относиться сгенерированные изображения

Преобразует метку желаемого класса в вектор длины latentSize посредством поиска вложения

Объединяет латентный вектор и условное вложение класса посредством умножения

Модель с последовательной сверточной сетью в качестве ядра готова

10.3.3. Детальнее исследуем обучение ACGAN

После чтения предыдущего раздела вы должны лучше понимать внутреннюю структуру дискриминатора и генератора ACGAN и включение в них информации о классе изображений цифр (часть AC названия ACGAN). С этими знаниями мы готовы развернуть рис. 10.10 и 10.11 и во всех деталях разобраться, как обучается ACGAN.

Рисунок 10.13 представляет собой расширенную версию рис. 10.10. На нем показано обучение дискриминатора ACGAN. По сравнению с предыдущим вариантом, этот шаг обучения не только повышает способность дискриминатора различать настоящие и сгенерированные (поддельные) изображения, но и оттачивает его возможности определения того, к какому классу цифры относится заданное изображение (настоящее или сгенерированное). Чтобы вам проще было сравнивать с предыдущей схемой, мы изобразили серым цветом части, которые вы уже видели на рис. 10.10, и выделили новые части.

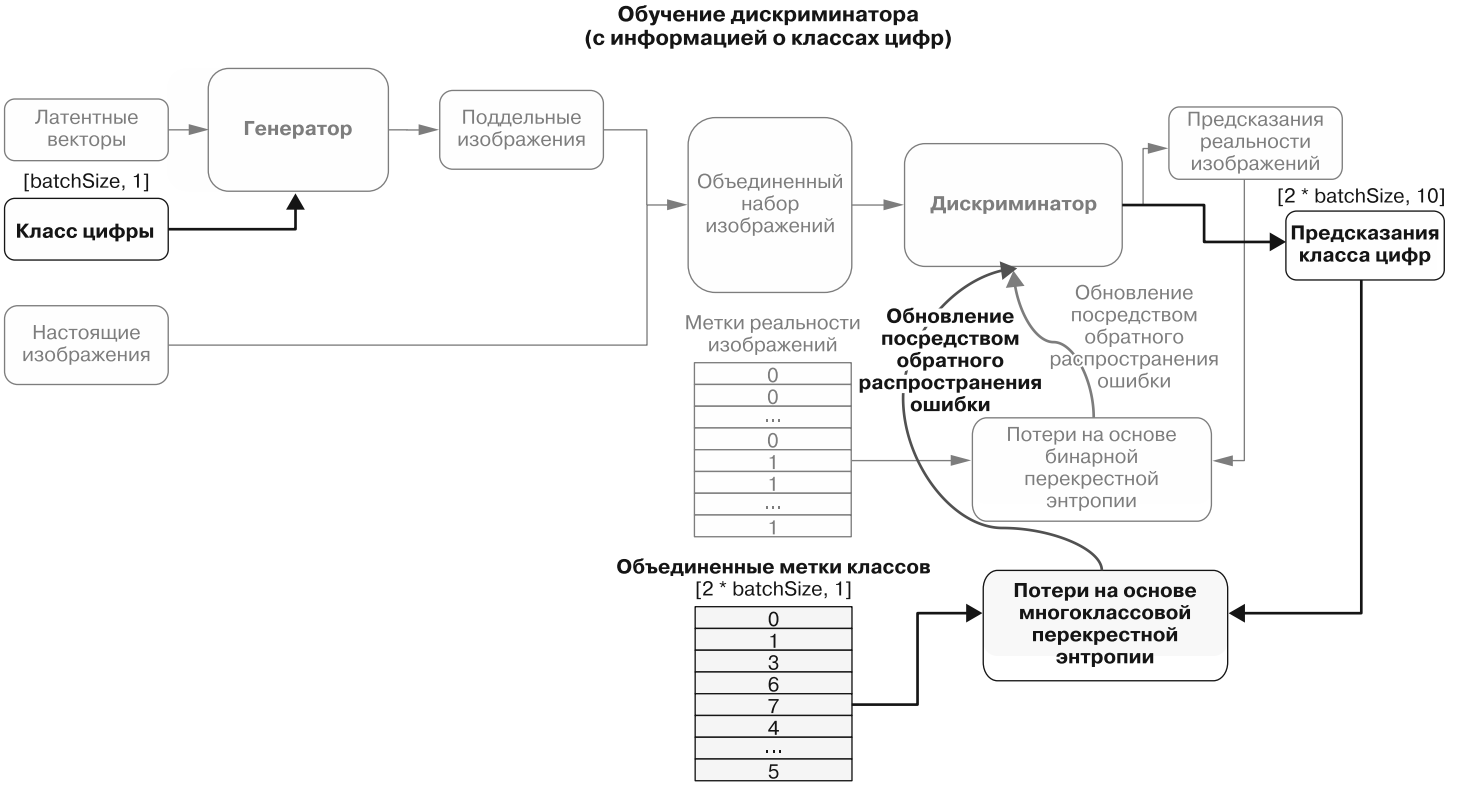


Рис. 10.13. Схематическая иллюстрация алгоритма обучения дискриминатора GAN. Дополняет схему с рис. 10.10 частями, относящимися к классам цифр. Прочие части схемы, приведенные на рис. 10.10, отображены светло-серым цветом

Во-первых, обратите внимание на дополнительный входной сигнал генератора (класс цифры), позволяющий указывать ему, изображения каких цифр генерировать. Кроме того, дискриминатор теперь выводит не только предсказание относительно подлинности изображения, но и предсказание класса цифры. В результате необходимо обучать обе выходные верхушки дискриминатора. Обучение части, предсказывающей реальность изображения, остается таким же, как и раньше (см. рис. 10.10); обучение части, предсказывающей класс цифры, основывается на том, что нам известно, к каким классам цифр относятся сгенерированные, а к каким — настоящие изображения. Эти две верхушки модели компилируются с различными функциями потерь, отражающими разную сущность предсказаний. Для предсказания подлинности мы используем в качестве функции потерь бинарную перекрестную энтропию, а для предсказания класса цифры — разреженную категориальную перекрестную энтропию. Это видно в следующей строке кода из файла `mnist-acgan/gan.js`:

```
discriminator.compile({
  optimizer: tf.train.adam(args.learningRate, args.adamBeta1),
  loss: ['binaryCrossentropy', 'sparseCategoricalCrossentropy']
});
```

Как демонстрируют две изогнутые стрелки на рис. 10.13, распространяющиеся обратно градиенты для обеих функций потерь прибавляются друг к другу при обновлении весовых коэффициентов дискриминатора. Рисунок 10.14 представляет собой расширенную версию рис. 10.11, на нем приведена подробная схема обучения генератора ACGAN. Эта схема показывает, как генератор обучается генерировать правильные изображения для конкретных классов цифр, помимо того, что обучается генерировать правдоподобные изображения. Как и на рис. 10.13, новые части выделены, а уже присутствовавшие на рис. 10.11 части отображены серым цветом. Из выделенных частей видно, что метки, подаваемые на вход шага обучения, теперь включают не только метки реальности изображений, но и метки классов цифр. Как и ранее, все метки реальности специально сделаны равными 1. Добавленные же новые метки классов цифр более «правдивы» в том смысле, что мы действительно пропустили их через генератор.

Ранее мы видели, как в результате любых расхождений между фиктивными метками реальности изображений и выходными значениями вероятностей дискриминатора генератор ACGAN обновляется так, что совершенствует свое умение «обманывать» дискриминатор. Метки классов цифр здесь играют схожую роль. Например, если мы сказали генератору сгенерировать изображение для цифры 8, но дискриминатор классифицировал его как изображение цифры 9, значение разреженной категориальной перекрестной энтропии будет высоким, как и значения соответствующих градиентов. В результате весовые коэффициенты генератора обновятся таким образом, что генератор в следующий раз сгенерирует изображение, более напоминающее цифру 8 (с точки зрения дискриминатора). Разумеется, такое обучение генератора сработает, только если дискриминатор уже достаточно хорошо умеет классифицировать изображения по десяти классам цифр MNIST. Это помогает обеспечить вышеописанный шаг обучения дискриминатора. Опять же мы видим, как дискриминатор и генератор сплетаются, подобно инь и ян, во время обучения ACGAN.

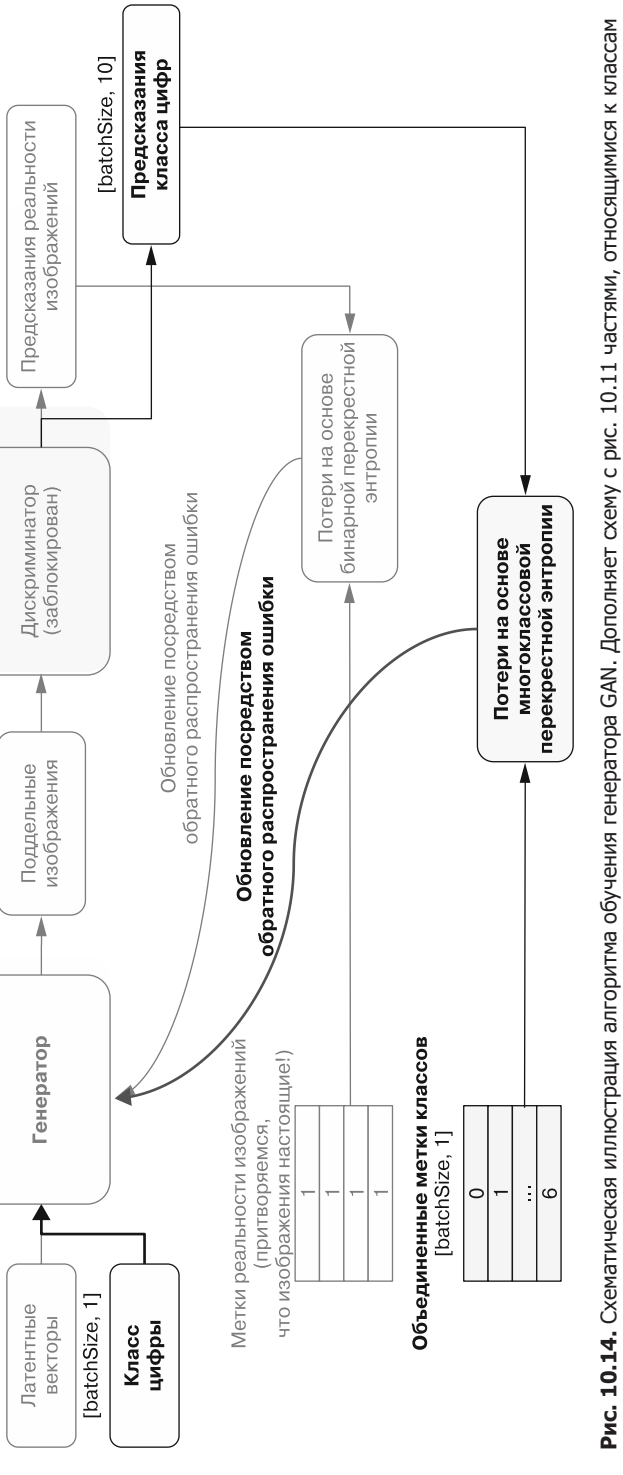


Рис. 10.14. Схематическая иллюстрация алгоритма обучения генератора GAN. Дополняет схему с рис. 10.11 частями, относящимися к классам цифр. Прочие части схемы, уже приведенные на рис. 10.11, отображены серым цветом

Обучение GAN: арсенал приемов

Процесс обучения и настройки GAN печально известен своей сложностью. Обучающие сценарии, приведенные в примере mnist-acgan, — результат колоссальной работы исследователей методом проб и ошибок. Как и многое другое в глубоком обучении, это скорее искусство, а не точная наука — эвристические правила, а не систематизированная теория. Они подтверждаются определенной степенью интуитивного понимания исследуемого феномена и достаточно хорошо работают на практике, хотя и не всегда.

Вот список заслуживающих упоминания приемов, которые мы использовали в ACGAN в этом разделе.

- В качестве функции активации последнего слоя conv2dTranspose в генераторе мы использовали th. В прочих типах моделей активация на основе th встречается реже.
- Повысить устойчивость к ошибкам помогает введение в модель элемента случайности. Поскольку обучение GAN порой приводит к динамическому равновесию, GAN способна завязнуть на одном месте самыми разнообразными способами. Введение элемента случайности во время обучения помогает предотвратить подобный исход. Мы будем вводить в модель случайность двумя способами: с помощью дропаута в дискриминаторе или «слабой единицы» (0,95) в качестве значений меток реальности в дискриминаторе.
- Создавать проблемы при обучении GAN могут разреженные градиенты (градиенты, существенная доля значений которых равна нулю). В прочих разновидностях глубокого обучения разреженность часто желательна, но не в GAN. Привести к разреженности градиентов могут субдискретизация с выбором максимального значения и функции активации ReLU. Поэтому вместо выбора максимального значения при субдискретизации рекомендуется использовать шаговую свертку (strided convolution), как мы и делаем в коде создания генератора в листинге 10.10. Вместо обычной функции активации ReLU рекомендуется использовать функцию активации leakyReLU с небольшим отрицательным значением у отрицательной части вместо нуля. Ее также можно видеть в листинге 10.10.

10.3.4. Обучение ACGAN для набора данных MNIST и генерация изображений в действии

Пример mnist-acgan можно извлечь и подготовить к запуску с помощью следующих команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/mnist-acgan
yarn
```

Выполнение примера включает два этапа: обучение в Node.js и генерацию в браузере. Для запуска процесса обучения достаточно такой команды:

```
yarn train
```

В процессе обучения по умолчанию используется tfjs-node. Впрочем, как и в предыдущих примерах сверточных сетей, скорость обучения можно значительно повысить за счет использования tfjs-node-gpu. Для этого при наличии настроенного должным образом GPU с поддержкой CUDA достаточно добавить в команду `yarn train` флаг `--gpu`. Обучение ACGAN займет не менее нескольких часов. Для мониторинга хода выполнения этого «долгоиграющего» задания можно воспользоваться TensorBoard, добавив флаг `--logDir`:

```
yarn train --logDir /tmp/mnist-acgan-logs
```

После открытия TensorBoard в отдельном терминале с помощью следующей команды:

```
tensorboard --logdir /tmp/mnist-acgan-logs
```

можно перейти по URL TensorBoard (выводится в консоль серверным процессом TensorBoard) в браузере и посмотреть на кривые потерь. На рис. 10.15 приведены примеры кривых потерь процесса обучения. Одна из характерных особенностей кривых потерь обучения GAN — они не обязательно стремятся вниз, подобно кривым потерь большинства других типов нейронных сетей. Как потери для дискриминатора (dLoss на рисунке), так и потери для генератора (gLoss на рисунке) меняются немонотонно и замысловато отражают друг друга.

Ни одна из функций потерь не приближается к нулю ближе к концу обучения, они просто устанавливаются на определенном уровне (сходятся). На этом этапе процесс обучения завершается, относящаяся к генератору часть модели сохраняется на диск для использования на шаге генерации в браузере:

```
await generator.save(saveURL);
```

Запустить демонстрацию генерации в браузере можно с помощью команды `yarn watch`. Эта команда компилирует файл `mnist-acgan/index.js` и соответствующие HTML- и CSS-ресурсы, после чего открывает вкладку в браузере и отображает страницу демонстрации[1].

Страница демонстрации загружает обученный генератор ACGAN, сохраненный на предыдущем этапе. Поскольку для текущего этапа дискриминатор особо не нужен, он не сохраняется и не загружается. После загрузки генератора можно сформировать батч латентных векторов вместе с батчем желаемых индексов классов цифр и вызвать с ними в качестве аргументов метод `predict()` генератора. Выполняющий эти действия код из файла `mnist-acgan/index.js`:

```
const latentVectors = getLatentVectors(10);
const sampledLabels = tf.tensor2d(
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [10, 1]);
const generatedImages =
    generator.predict([latentVectors, sampledLabels]).add(1).div(2);
```

[1] Можете полностью пропустить шаги обучения и сборки и перейти сразу к странице демонстрации, размещенной по адресу http://mng.bz/4eGw.

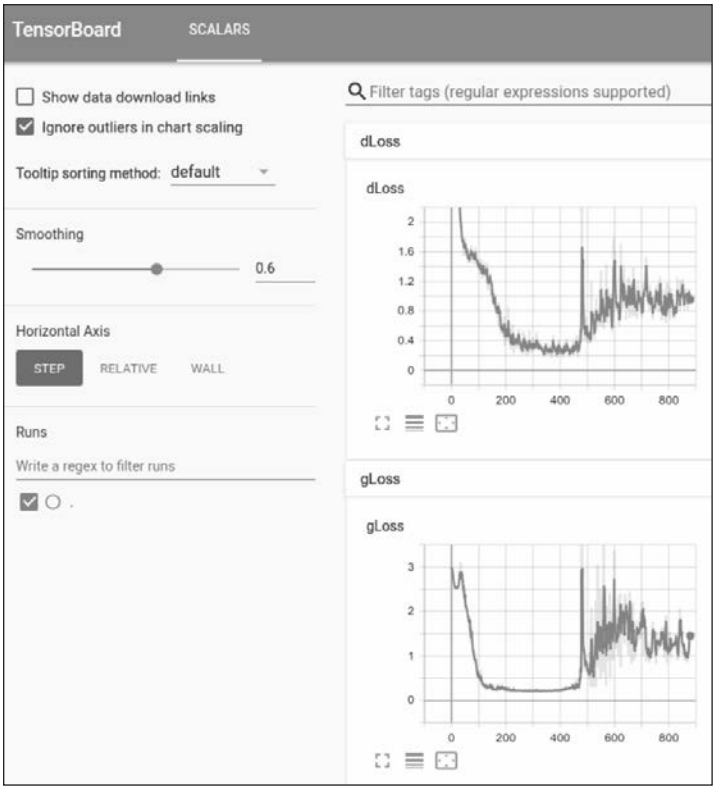


Рис. 10.15. Примеры кривых потерь процесса обучения ACGAN. dLoss — потери с шага обучения дискриминатора. Если точнее, они представляют собой сумму бинарной перекрестной энтропии от предсказания подлинности изображений и разреженной категориальной перекрестной энтропии от предсказания классов цифр. gLoss — потери с шага обучения генератора. Подобно dLoss, gLoss равны сумме потерь от бинарной классификации подлинности и многоклассовой классификации цифр

Батч меток классов цифр всегда представляет собой упорядоченный вектор из десяти элементов, от 0 до 9, поэтому батч сгенерированных изображений всегда представляет собой упорядоченный массив изображений от 0 до 9. Эти изображения соединяются с помощью функции `tf.concat()` и выводятся в элементе `div` на веб-странице (рис. 10.16, *вверху*). При сравнении с выбранными случайным образом настоящими изображениями MNIST (см. рис. 10.16, *внизу*) видно, что эти сгенерированные ACGAN изображения выглядят как настоящие. Кроме того, их классы цифр правильны, что демонстрирует успешность обучения ACGAN. Чтобы получить больше выходных изображений от генератора ACGAN, нажмите на странице кнопку Generator. При каждом нажатии этой кнопки генерируется и выводится на странице новый батч из десяти фальшивых изображений. Можете поэкспериментировать с этой возможностью, чтобы лучше прочувствовать качество генерации изображений.

Рис. 10.16. Пример изображений (верхний блок 10 × 1), сгенерированных генератором обученного ACGAN. Для сравнения показан нижний блок, содержащий таблицу 10 × 10 настоящих изображений из набора данных MNIST. При нажатии кнопки Show Z-vector Sliders открывается область, заполненная 100 слайдерами, с помощью которых можно поменять элементы латентного вектора (z-вектора) и посмотреть, что поменяется в сгенерированных изображениях MNIST. Учтите, что влияние на изображения большинства слайдеров при изменении их по одному очень незначительно и малозаметно. Но иногда можно случайно натолкнуться на слайдер с более заметным эффектом

Материалы для дальнейшего изучения

- *Goodfellow I., Bengio Y., Courville A.* Deep Generative Models // Deep Learning, chapter 20, MIT Press, 2017. (*Гудфеллоу Я., Бенджио Б., Курвилль А.* Глубокое обучение. — М.: ДМК-Пресс, 2018; глава 20 «Глубокие порождающие модели».)
- *Langr J., Bok V.* GANs in Action: Deep Learning with Generative Adversarial Networks, Manning Publications, 2019.
- *Karpathy A.* The Unreasonable Effectiveness of Recurrent Neural Networks // blog, 21 May 2015: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.
- *Hui J.* GAN — What is Generative Adversary Networks GAN? // Medium, 19 June 2018: http://mng.bz/Q0N6.
- GAN Lab — интерактивная веб-среда для исследования работы GAN, созданная с помощью TensorFlow.js: *Kahng M. et al.:* https://poloclub.github.io/ganlab/.

Упражнения

1. Помимо корпуса шекспировских текстов, в примере lstm-text-generation есть несколько других настроенных и готовых к изучению наборов текстов. Запустите для них процесс обучения и посмотрите, что получится. Например, в качестве обучающего набора данных воспользуйтесь полным кодом TensorFlow.js. Во время и после обучения модели понаблюдайте, демонстрирует ли сгенерированный текст следующие закономерности исходного кода JavaScript, а также влияние на них параметра температуры.
   А. «Короткие» паттерны, такие как ключевые слова (например, `for` и `function`).
   Б. «Средние» паттерны наподобие построчной организации кода.
   В. «Далекие» паттерны, такие как попарное соответствие круглых и квадратных скобок, а также тот факт, что за каждым ключевым словом `function` должны следовать пара скобок и пара фигурных скобок.
2. Что получится, если убрать в примере fashion-mnist-vae компонент расстояния Кульбака — Лейблера из нашей пользовательской функции потерь VAE? Проверьте это, модифицировав функцию `vaeLoss()` в файле `fashion-mnist-vae/model.js` (см. листинг 10.7). Напоминают ли изображения, выбранные из латентного пространства, изображения из набора данных Fashion-MNIST? Демонстрирует ли по-прежнему это пространство какие-то понятные закономерности?
3. Попробуйте схлопнуть в примере mnist-acgan десять классов цифр в пять (первый класс состоит из 0 и 1, второй — из 2 и 3 и т. д.) и посмотрите, как поменяется выходной сигнал ACGAN после обучения. Что вы ожидаете увидеть в сгенерированных изображениях? Например, как вы думаете, что сгенерирует ACGAN, если потребовать сгенерировать изображение первого класса?

   Подсказка: для этого вам понадобится внести изменения в функцию `loadLabels()` из файла `mnist-acgan/data.js`. Нужно также поменять соответствующим образом константу `NUM_CLASSES` в файле `gan.js`. Кроме того, необходимо модифицировать переменную `sampledLabels` в функции `generateAndVisualizeImages()` (в файле `index.js`).

Резюме

- Генеративные модели отличаются от дискриминативных, изученных в предыдущих главах, моделированием процесса, в котором генерируются примеры обучающего набора данных вместе с их статистическими распределениями. Благодаря такой архитектуре они могут генерировать новые примеры данных, удовлетворяющие нужным распределениям, а потому напоминающие настоящие обучающие данные.
- Мы познакомили вас со способом моделирования структуры текстовых наборов данных: предсказанием следующего символа. Для итеративного выполнения этой

задачи и генерации текста произвольной длины можно использовать LSTM. Стохастичностью (степенью случайности и непредсказуемости) сгенерированного текста можно управлять с помощью параметра температуры.

- Автокодировщик — тип генеративной модели, состоящий из кодировщика и декодировщика. Сначала кодировщик уплотняет входные данные в сжатое представление — латентный вектор (z-вектор). Затем декодировщик пытается восстановить исходные данные по одному латентному вектору. В процессе обучения кодировщик становится высокоэффективным средством сжатия данных, а декодировщик наделяется знанием статистического распределения примеров данных. VAE накладывает дополнительные статистические ограничения на латентные векторы так, что содержащие эти векторы латентные пространства после обучения VAE демонстрируют непрерывно меняющиеся и удобные для интерпретации структуры.
- В основе GAN лежит идея «единства и борьбы» дискриминатора и генератора. Дискриминатор пытается отличить настоящие примеры данных от сгенерированных, а генератор стремится сгенерировать поддельные примеры, способные «обмануть» дискриминатор. В ходе их совместного обучения генератор постепенно становится способен генерировать правдоподобные примеры данных. ACGAN добавляет в базовую архитектуру GAN информацию о классах, позволяя задавать желаемый класс примеров данных для генерации.

11 Основы глубокого обучения с подкреплением

В этой главе

- Отличия обучения с подкреплением от обсуждавшегося в предыдущих главах обучения с учителем.
- Основная парадигма обучения с подкреплением: агент, среда, действие и вознаграждение, а также их взаимодействия.
- Основные идеи двух главных подходов к решению задач обучения с подкреплением: на основе стратегий и на основе ценности.

До сих пор в книге мы сосредотачивали свое внимание в основном на так называемом *обучении с учителем* (supervised learning). При таком обучении модель обучается правильно реагировать на входной сигнал. Присваивает ли она метку класса входному изображению (см. главу 4) или предсказывает температуру в будущем по прошлым метеорологическим данным (см. главы 8 и 9), парадигма остается неизменной: статическим входным данным ставятся в соответствие статические выходные данные. Модели генерации последовательностей, обсуждавшиеся в главах 9 и 10, были чуть сложнее, поскольку выходной сигнал в них представлял собой последовательность элементов, а не отдельный элемент. Но соответствующие задачи все равно можно свести к варианту «один элемент на входе — один элемент на выходе», разбив последовательности на шаги.

В этой главе мы рассмотрим совершенно иную разновидность машинного обучения — *обучение с подкреплением* (reinforcement learning, RL). В нем главное

не статический выходной сигнал; модель (*агент* (agent) в терминологии RL) обучается воздействовать на среду так, чтобы максимизировать метрику успешности — *вознаграждение* (reward). Например, обучение с подкреплением можно использовать, чтобы научить робота обходить внутренние помещения здания и собирать мусор. На самом деле среда не обязательно должна быть физической, ею может быть любое реальное или виртуальное пространство, в котором агент выполняет какие-либо действия. Средой, на которой агента можно обучить играть в шахматы, служит шахматная доска; биржа может послужить средой для обучения агента торговле акциями. Универсальность парадигмы обучения с подкреплением позволяет применять его к широкому спектру практических задач (рис. 11.1). Кроме того, объединению возможностей глубокого обучения с RL мы обязаны некоторыми наиболее впечатляющими достижениями революции глубокого обучения, включая чат-боты, играющие в Atari со сверхчеловеческим мастерством, и алгоритмы, способные побеждать чемпионов мира в го и шахматы[1].

Рис. 11.1. Примеры реальных приложений обучения с подкреплением. *Слева вверху:* Решение задач настольных игр, в частности го и шахмат. *Справа вверху:* алгоритмическая торговля акциями. *Слева внизу:* автоматизированное управление ресурсами в центрах обработки данных. *Справа внизу:* планирование управления и действий в робототехнике. Все изображения лицензированы как свободно доступные, скачаны с сайта http://www.pexels.com/

[1] *Silver D. et al.* Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm // submitted 5 Dec. 2017. https://arxiv.org/abs/1712.01815.

Захватывающая область обучения с подкреплением коренным образом отличается от задач обучения с учителем из предыдущих глав. Задача RL состоит в поиске оптимальных процессов принятия решений путем взаимодействия со средой, в отличие от усвоения соответствий входных и выходных сигналов, как в обучении с учителем. В RL у нас нет маркированных обучающих наборов данных, лишь различные типы сред для исследования. Кроме того, неотъемлемым основополагающим измерением в задачах RL является время, в отличие от многих задач обучения с учителем, где измерение времени либо отсутствует, либо играет роль фактически еще одного пространственного измерения. Вследствие уникальных характеристик обучения с подкреплением терминология и образ мыслей в этой главе будет сильно отличаться от того, что вы прочли в предыдущих главах. Но не беспокойтесь. Основные понятия и подходы мы проиллюстрируем на простых конкретных примерах. Кроме того, нам по-прежнему будут сопутствовать наши старые добрые друзья, глубокие нейронные сети и их реализации в TensorFlow.js, образуя один (хотя и не единственный!) из столпов алгоритмов обучения с подкреплением в этой главе.

К концу главы вы познакомитесь с постановкой основных задач RL, разберетесь с основными идеями двух чаще всего используемых в RL нейронных сетей (сетей стратегий и Q-сетей), а также научитесь обучать подобные сети с помощью API TensorFlow.js.

11.1. Постановка задач обучения с подкреплением

На рис. 11.2 приведены основные компоненты задачи обучения с подкреплением. Мы (специалисты по RL) можем непосредственно контролировать агента. Агент (например, робот, собирающий мусор в здании) взаимодействует со средой тремя способами.

- На каждом шаге агент выполняет *действие* (action), меняющее состояние среды. Применительно к нашему роботу, например, набор возможных действий может выглядеть так: `{переместиться вперед, переместиться назад, повернуть налево, повернуть направо, поднять мусор, сбросить мусор в контейнер}`.
- Изредка среда *вознаграждает* агент. Применительно к людям это вознаграждение можно сравнить с мерой одномоментного удовольствия или счастья. На более абстрактном языке вознаграждение (а точнее, сумма вознаграждений за определенный промежуток времени, как мы увидим далее) — это число, которое агент стремится максимизировать. Оно представляет собой важную числовую величину, направляющую ход алгоритмов обучения с подкреплением, подобно тому как величина потерь направляет ход алгоритмов обучения с учителем. Вознаграждение может быть положительным или отрицательным. Например, наш робот, собирающий мусор, может получать положительное вознаграждение при успешном сбросе порции мусора в контейнер. А отрицательное вознаграждение — если случайно опрокинет мусорное ведро, натолкнется на людей или мебель либо промахнется мимо контейнера при сбросе мусора.

- Помимо вознаграждения, агент может получать информацию о состоянии среды через другой канал, а именно *наблюдение* (observation). Видимым агенту может быть полное состояние среды или только его часть, возможно искаженная из-за низкого качества канала. Наблюдения нашего собирающего мусор робота представляют собой потоки изображений и сигналов с камер и различных датчиков на его корпусе.

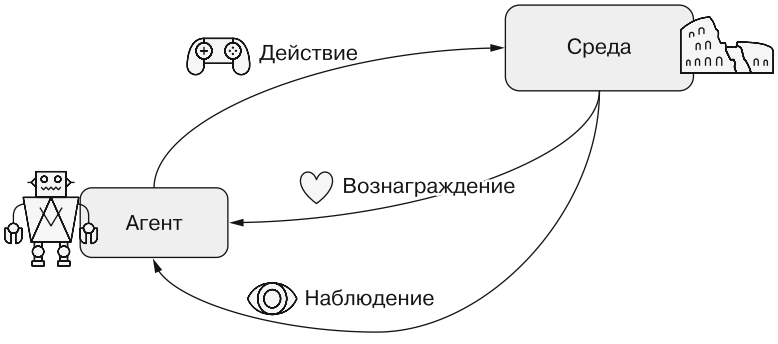


Рис. 11.2. Схематическая иллюстрация простейшей постановки задач обучения с подкреплением. На каждом шаге агент выбирает одно из возможных действий, меняющих состояние среды. Среда вознаграждает агент в соответствии с текущим состоянием и выбранным действием. Агент полностью или частично наблюдает состояние среды и использует полученную информацию для принятия решений о дальнейших действиях

Только что описанная постановка задачи несколько абстрактна. Взглянем на конкретные примеры задач обучения с подкреплением, чтобы прочувствовать спектр охватываемых этой формулировкой возможностей. В процессе этого мы также приведем классификацию всех существующих задач RL. Во-первых, рассмотрим действия. Пространство, из которого агент выбирает действия, может быть дискретным или непрерывным. Например, пространства действий у агентов RL, предназначенных для настольных игр, обычно дискретные в силу конечности возможных ходов. Однако задача RL управления ходьбой на двух ногах виртуального человекоподобного робота[1] требует непрерывного пространства действий, поскольку моменты сил в его шарнирах представляют собой непрерывные величины. Пространства действий в примерах задач, которые мы рассмотрим в этой главе, дискретны. Учтите, что в некоторых задачах RL непрерывные пространства действий можно превратить в дискретные посредством дискретизации. Например, агент для игры StarCraft II компании DeepMind разбивает двумерный экран в высоком разрешении на прямоугольники, чтобы определить, куда двигать юниты и где начинать атаки[2].

---

[1] См. среду Humanoid OpenAI по адресу https://gym.openai.com/envs/Humanoid-v2/.

[2] *Vinyals O. et al.* StarCraft II: A New Challenge for Reinforcement Learning // submitted 16 Aug. 2017. https://arxiv.org/abs/1708.04782.

Вознаграждения, играющие ключевую роль в задачах RL, также бывают разные. Во-первых, в некоторых задачах RL вознаграждение бывает только положительным. Например, как мы вскоре увидим, агент RL, задача которого состоит в удержании шеста в равновесии в движущейся тележке, получает лишь положительное вознаграждение. Он получает небольшое положительное вознаграждение за каждый временной шаг, на котором шест продолжает стоять. Однако во многих задачах RL встречается сочетание положительного и отрицательного вознаграждения. Отрицательное вознаграждение можно считать своего рода штрафом или наказанием. Например, агент, обучающийся забрасывать мяч в баскетбольное кольцо, должен получать положительное вознаграждение за заброшенные мячи и отрицательное — за промахи.

Вознаграждения могут различаться и частотой. Некоторые задачи RL характеризуются непрерывным потоком вознаграждений. В их числе вышеупомянутая задача удержания шеста в равновесии, например: пока шест стоит, агент получает (положительное) вознаграждение на каждом временном шаге. С другой стороны, рассмотрим агент RL для игры в шахматы: он получает вознаграждение только в конце, когда становится известен исход игры (победа, проигрыш или ничья). Существуют и промежуточные между этими двумя крайними случаями задачи RL. Например, наш робот для сборки мусора может не получать никакого вознаграждения на всех шагах между двумя успешными сбросами мусора в контейнер, то есть когда он просто перемещается из точки А в точку Б. Аналогично агент RL для игры Pong Atari не получает вознаграждения на каждом шаге (кадре) компьютерной игры, а вознаграждается положительно каждые несколько шагов, когда управляемая им бита ударяет по мячу и тот отскакивает к противнику. Среди примеров задач RL этой главы встречаются задачи как с высокой, так и с низкой частотой вознаграждения.

Наблюдение — еще один важный фактор в задачах обучения с подкреплением, своего рода окно, через которое агент может посматривать на состояние среды, формируя фундамент (помимо вознаграждений) для принятия решений. Как и действия, наблюдения могут быть дискретными (как в настольной или карточной игре) или непрерывными (как в физической среде). У вас может возникнуть вопрос: почему в нашей формулировке задач RL наблюдение и вознаграждение — две отдельные сущности, хотя и то и другое можно считать обратной связью от среды агенту. Дело в том, что это принципиально упрощает задачу и повышает ее понятность. Хотя вознаграждение можно считать разновидностью наблюдения, именно оно в конечном счете главное для агента. Наблюдение может включать как относящуюся, так и не относящуюся к делу информацию, которую агент должен уметь фильтровать и использовать с умом.

В одних задачах RL агент через наблюдение получает доступ ко всему состоянию среды, а в других — лишь к частям состояния. Примеры задач первого типа включают настольные игры, например шахматы и го. Хороший пример задач второго типа — карточные игры, такие как покер, в которых карты на руках противника неизвестны, а также торговля акциями. Цены на акции определяет множество факторов, например, внутренняя деятельность компаний и образ мыслей других

биржевых маклеров на рынке. Но агент может непосредственно наблюдать лишь малую толику этих состояний. В результате наблюдения агента ограничиваются поминутной историей цен на акции, возможно, в дополнение к общедоступной информации, такой как финансовые новости.

Это обсуждение определяет площадку, на которой происходит обучение с подкреплением. Стоит отметить интересный нюанс этой постановки задачи — двунаправленность потока информации между агентом и средой: агент воздействует на среду, а среда, в свою очередь, предоставляет агенту вознаграждения и информацию о состоянии. Данный нюанс отличает обучение с подкреплением от обучения с учителем, в котором поток информации в основном идет в одну сторону: входной сигнал содержит достаточно информации для предсказания алгоритмом выходного сигнала, выходной же сигнал никак особенно не воздействует на входной.

Еще одна интересная и уникальная особенность задач RL: они обязательно происходят вдоль измерения времени, для многоэтапности/многошаговости взаимодействий «агент — среда». Время может измеряться дискретно или непрерывно. Например, агенты RL для настольных игр обычно оперируют на непрерывной оси времени, поскольку эти игры состоят из отдельных ходов. То же самое относится к компьютерным играм. А вот ось времени для агента RL, управляющего механической рукой-манипулятором, должна быть непрерывной, хотя он может и выполнять действия в дискретные моменты времени. В этой главе мы сосредоточим свое внимание на задачах RL с дискретной осью времени.

Пока приведенных теоретических сведений об RL достаточно. В следующем разделе мы приступим к изучению реальных задач и алгоритмов обучения с подкреплением на практике.

11.2. Сети стратегий и градиентный спуск по стратегиям: пример cart-pole

Первая задача RL, которой мы займемся: моделирование механической системы, в которой тележка с установленным на ней шестом перемещается по прямолинейной дорожке. Эта задача была впервые описана Эндрю Барто, Ричардом Саттоном и Чарльзом Андерсоном в 1983 году[1] и уже стала эталонной в сфере проектирования систем управления (подобно задаче распознавания цифр MNIST для обучения с учителем) благодаря своей простоте и четкости математической и физической формулировки наряду с относительной сложностью решения. Здесь задача агента — управлять движением тележки путем приложения направленных вправо или влево сил, чтобы удержать шест в равновесии настолько долго, насколько это возможно.

---

[1] *Barto A. G., Sutton R. S., Anderson C. W.* Neuronlike Adaptive Elements that Can Solve Difficult Learning Control Problems // IEEE Transactions on Systems, Man, and Cybernetics, Sept./Oct. 1983. Pp. 834–846. http://mng.bz/Q0rG.

11.2.1. Удержание шеста в равновесии в тележке как задача обучения с подкреплением

Прежде чем продолжать, рекомендуем вам поэкспериментировать с примером cart-pole, чтобы хорошенько прочувствовать задачу. Она достаточно проста для моделирования и обучения модели в браузере. На рис. 11.3 задача удержания равновесия шеста в тележке изображена наглядно, аналогичную картинку вы можете найти на странице, которая откроется после выполнения команды `yarn watch`. Для извлечения и запуска примера выполните следующие команды:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/cart-pole
yarn && yarn watch
```

Нажмите кнопку Create Model (Создать модель), а затем кнопку Train (Обучить). Внизу страницы появится анимация, демонстрирующая необученный агент, удерживающий шест в равновесии в тележке. Поскольку весовые коэффициенты модели агента были инициализированы случайными значениями (больше о модели мы расскажем позднее), сначала он не слишком хорошо решает эту задачу. В терминологии обучения с подкреплением все временные шаги от начала до конца игры иногда в совокупности называются *эпизодом* (episode). Мы будем использовать здесь попеременно термины *«игра»* и *«эпизод»*.

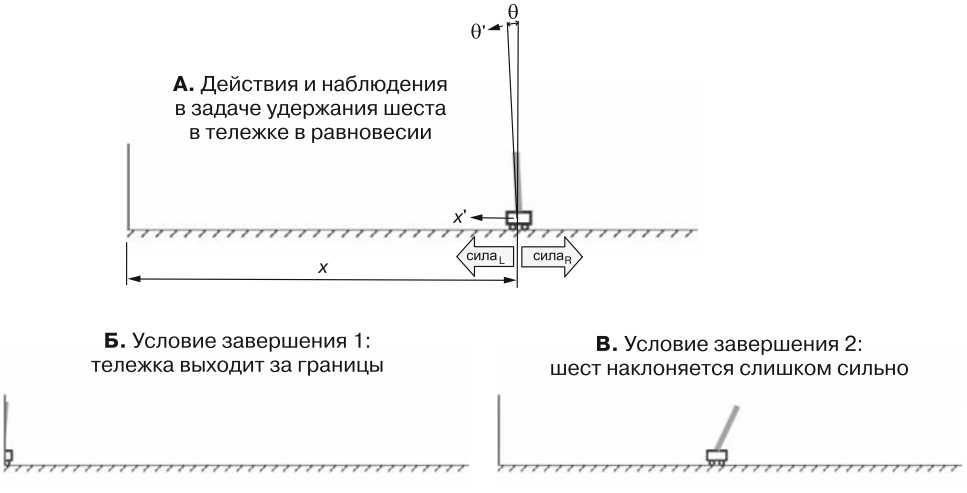


Рис. 11.3. Наглядная иллюстрация задачи по удержанию равновесия шеста в тележке. Блок А: состояние среды и наблюдение составляют четыре физические величины (местоположение тележки *x*, скорость тележки *x'*, угол наклона шеста θ и угловая скорость шеста θ'). На каждом временном шаге агент может выбрать, приложить ли ему направленную вправо или влево силу, в результате чего состояние среды поменяется соответствующим образом. Блоки Б и В: два условия завершения игры — выход тележки за правую/левую границу (Б) или слишком сильный наклон шеста от вертикального положения (В)

Как демонстрирует блок А на рис. 11.3, переменная x в каждый момент времени захватывает местоположение тележки на рельсах. Мгновенная скорость тележки обозначается x'. Кроме того, еще одна переменная, θ, захватывает угол наклона шеста. Угловая скорость шеста (как изменяется θ и в каком направлении) обозначается θ'. Все вместе эти четыре физические величины (x, x', θ и θ') полностью наблюдаются агентом на каждом шаге и составляют наблюдение данной задачи RL.

Моделирование завершается при выполнении одного из двух условий.

- Значение x выходит за заданные пределы, или в физическом смысле тележка наталкивается на одну из стен по сторонам рельсов (блок Б на рис. 11.3).
- Модуль θ превышает определенное пороговое значение, или в физическом смысле шест слишком сильно отклоняется от вертикального положения (блок В на рис. 11.3).

Среда также завершает эпизод после 500-го шага моделирования, предотвращая слишком длительную игру (что случается, когда агент научился слишком хорошо играть). Верхнюю границу количества шагов можно настраивать в UI. Агент получает единичное вознаграждение (`1`) на каждом шаге моделирования вплоть до завершения игры. Таким образом, чтобы получить максимальное суммарное вознаграждение, агент должен найти способ продержать шест вертикально как можно дольше. Но каким образом он управляет системой «тележка — шест»? Тут-то мы и сталкиваемся с действиями в данной задаче RL.

Как показывают стрелки «Сила» в блоке А на рис. 11.3, агент на каждом ходе ограничен двумя возможными действиями: приложение силы к тележке вправо или влево. Агент должен выбрать одно из этих двух направлений приложения силы. Величина прикладываемой силы фиксирована. После приложения силы модель использует ряд математических уравнений для вычисления следующего состояния (новых значений x', x', θ и θ') среды, описывающих классическую механику Ньютона. Мы не станем углубляться в подробности этих уравнений, поскольку они для нас неважны, но, если вам интересно, их можно найти в файле `cart-pole/cart_pole.js`.

Аналогично код визуализации системы «тележка — шест» на HTML-холсте можно найти в файле `cart-pole/ui.js`. Он подчеркивает преимущества написания алгоритмов RL на языке JavaScript (в частности, на TensorFlow.js): возможность написания UI и алгоритмов обучения на одном языке и тесной интеграции их друг с другом. Что, в свою очередь, упрощает визуализацию и повышает интуитивную понятность задачи, а также ускоряет процесс разработки. Резюмируем задачу удержания равновесия шеста в тележке, описав ее в канонической постановке RL (табл. 11.1).

Таблица 11.1. Задача удержания равновесия шеста в тележке в канонической постановке RL

Абстрактное понятие RL	**Воплощение в задаче удержания равновесия шеста в тележке**
Среда	Тележка с шестом, которая перемещается по прямолинейной дорожке
Действие	(Дискретное.) Бинарный выбор на каждом ходе между направленной влево и направленной вправо силой. Величина силы фиксирована

Абстрактное понятие RL	Воплощение в задаче удержания равновесия шеста в тележке
Вознаграждение	(Частое и исключительно положительное.) На каждом ходе эпизода игры агент получает фиксированное вознаграждение (1). Эпизод завершается, когда тележка ударяется о стену с одной из сторон дорожки или когда шест слишком сильно отклоняется от вертикального направления
Наблюдение	(Полное состояние, непрерывное.) На каждом ходе агент обладает доступом к полному состоянию системы «тележка — шест», включая местоположение тележки (x), ее скорость (x'), помимо угла наклона шеста (θ) и угловой скорости шеста (θ')

11.2.2. Сети стратегий

Мы сформулировали задачу и можем теперь приступить к ее решению. В прошлом специалисты по теории управления придумывали разнообразные хитроумные решения этой задачи, основанные на механике системы[1]. Но мы будем решать ее *не* так. В контексте книги подобный подход был бы подобен написанию эвристических правил анализа границ и углов изображений MNIST для классификации цифр. Вместо этого мы проигнорируем физику системы, наш агент будет обучаться методом проб и ошибок в полном соответствии с духом остальной части книги: вместо того чтобы жестко «зашивать» в код алгоритм или вручную проектировать признаки на основе знаний человека, мы спроектируем алгоритм, с помощью которого модель сможет обучаться самостоятельно.

Как сделать так, чтобы агент мог выбирать действие (приложить силу вправо или влево) на каждом ходе? При доступных агенту наблюдениях и решении, которое он должен принимать на каждом ходе, можно переформулировать задачу в виде простого задания отображать входные данные в выходные, аналогичного уже встречавшимся нам в обучении с учителем. Естественным решением будет нейронная сеть для выбора действия на основе наблюдения. Эта идея и лежит в основе так называемых *сетей стратегий* (policy networks).

Подобная нейронная сеть принимает на входе вектор наблюдений длиной 4 (x, x', θ и θ') и выдает на выходе число, которое можно истолковать как выбор «право — лево». Архитектура этой сети аналогична бинарному классификатору для обнаружения фишинговых сайтов в главе 3. Говоря абстрактным языком, на каждом шаге мы на основе информации о среде выбираем с помощью сети действие. Благодаря отыгрышу нескольких партий сеть собирает информацию для оценки принимаемых решений. Далее нам понадобится способ измерить качество этих решений, чтобы подогнать весовые коэффициенты сети. Тогда она в будущем сможет принимать решения, более похожие на «хорошие» и менее похожие на «плохие».

[1] Если вам интересны традиционные, без использования RL, решения задачи балансировки шеста на тележке и сложная математика вас не пугает, можете прочитать общедоступный курс по теории управления MIT Расса Тидрейка: http://mng.bz/j5lp.

Основные отличия этой системы от вышеприведенного классификатора.

- За время игрового эпизода модель вызывается несколько раз (на каждом временном шаге).
- Выходной сигнал модели (выходной сигнал из прямоугольника «Сеть стратегий» на рис. 11.4) представляет собой логиты, а не оценки вероятности. Далее логиты преобразуются в оценки вероятности с помощью сигма-функции. Но мы не включаем нелинейность в виде сигма-функции в последний (выходной) слой сети стратегий, поскольку для обучения нам понадобятся логиты, как вы скоро увидите.
- Вероятности на выходе сигма-функции необходимо преобразовать в конкретное действие (приложение силы вправо/влево). Для этого служит вызов функции `tf.multinomial()`, производящей случайную выборку. Как вы помните, мы использовали функцию `tf.multinomial()` в главе 10, при выборке следующего символа на основе вероятностей символов алфавита из многомерной логистической функции. Сейчас все немного проще, поскольку необходимо выбрать один из всего двух вариантов.

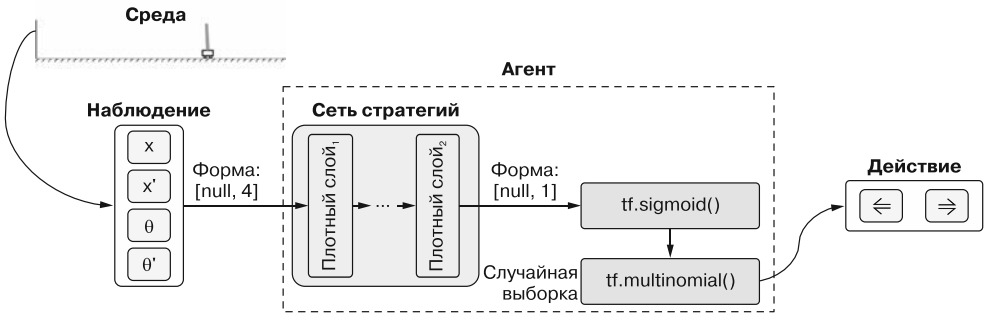


Рис. 11.4. Решение задачи удержания равновесия шеста на тележке с помощью сети стратегий. Данная сеть стратегий представляет собой модель TensorFlow.js, выдающую на выходе вероятность действия «приложение силы влево» при векторе наблюдений (x, x', θ и θ') в качестве входного сигнала. Вероятность преобразуется в нужное действие путем случайной выборки

Последний из этих пунктов влечет за собой более серьезные последствия. Мы ведь *могли* бы преобразовать выходной сигнал функции `tf.sigmoid()` непосредственно в действие с помощью порогового значения (например, выбирать действие приложения силы влево, когда выходной сигнал сети больше 0,5 и вправо в противном случае). Почему же мы предпочли этому простому подходу более сложный вариант со случайной выборкой с помощью функции `tf.multinomial()`? Дело в том, что нам *нужна* стохастичность функции `tf.multinomial()`. В самом начале обучения сеть стратегий не имеет понятия, как выбрать направление приложения силы, поскольку начальные значения весов задаются случайным образом. Благодаря случайной выборке сеть поощряется выполнять случайные действия и находить среди них те, что обеспечивают наилучшие результаты. Одна часть этих случайных проб демонстрирует не слишком хорошие результаты, а другая — хорошие. Наш алгоритм

запоминает удачные варианты и чаще использует их в будущем. Но найти такие удачные варианты не удастся, если не разрешить агенту пробовать случайным образом. При выборе детерминированного подхода с пороговым значением модель «застрянет» в самом начале.

Это приводит нас к классическому и очень важному вопросу RL: *исследовать или использовать* (exploration versus exploitation)? Под *исследованием* в этом контексте понимаются случайные пробы, с его помощью агент RL находит удачные варианты действий. Под *использованием* понимаются усвоенные агентом оптимальные решения, максимизирующие вознаграждение. Эти два режима работы несовместимы друг с другом. Поиск равновесия между ними критически важен для создания хорошо работающих алгоритмов обучения с подкреплением. Сначала необходимо исследовать широкий спектр возможных стратегий, но по мере схождения алгоритма к лучшим стратегиям нужно реализовать более точный их подбор. Поэтому по мере обучения во многих алгоритмах степень исследования обычно постепенно сокращается. В задаче удержания равновесия шеста на тележке такое сокращение производится функцией выборки `tf.multinomial()` неявно, в результате все большей детерминированности исходов по мере роста уровня уверенности модели в ходе обучения.

Листинг 11.1 (фрагмент из `cart-pole/index.js`) демонстрирует вызовы TensorFlow.js, создающие сеть стратегий. Код в листинге 11.2 (также фрагмент из `cart-pole/index.js`) преобразует выходной сигнал сети стратегий в действия агента, а также возвращает логиты для целей обучения. Связанный с моделью код здесь не слишком отличается от кода моделей обучения с учителем в предыдущих главах.

Листинг 11.1. Сеть стратегий MLP: выбираем действия на основе наблюдений

```
createModel(hiddenLayerSizes) {
    if (!Array.isArray(hiddenLayerSizes)) {
      hiddenLayerSizes = [hiddenLayerSizes];
    }
    this.model = tf.sequential();
    hiddenLayerSizes.forEach((hiddenLayerSize, i) => {
      this.model.add(tf.layers.dense({
        units: hiddenLayerSize,
        activation: 'elu',
        inputShape: i === 0 ? [4] : undefined
      }));
    });
    this.model.add(tf.layers.dense({units: 1}));
  }
}
```

Параметр hiddenLayerSizes определяет размеры всех слоев сети стратегий, за исключением последнего (выходного)

Параметр inputShape необходим только для первого слоя

Количество нейронов в последнем слое жестко задано в коде: 1. Входной сигнал из одного числа преобразуется далее в вероятность выбора приложения силы влево

Принципиальное отличие состоит в отсутствии здесь набора маркированных данных, которые можно было бы использовать, чтобы обучить модель тому, какие варианты выбираемых действий удачны, а какие — нет. При наличии подобного набора данных для решения задачи можно было бы просто вызвать `fit()` и `fitDataset()` сети стратегий, как мы делали для моделей в предыдущих главах. Но раз такого набора у нас нет, агент должен сам выяснять, какие действия удачные, играя в игру

и анализируя получаемые вознаграждения. Другими словами, он должен «учиться плавать, плавая» — ключевая особенность задач RL. Далее мы во всех подробностях рассмотрим, как это происходит.

Листинг 11.2. Определение логитов и действий по выходному сигналу сети стратегий

```
getLogitsAndActions(inputs) {
  return tf.tidy(() => {
    const logits = this.policyNet.predict(inputs);
    const leftProb = tf.sigmoid(logits);
    const leftRightProbs = tf.concat(
        [leftProb, tf.sub(1, leftProb)], 1);
    const actions = tf.multinomial(
        leftRightProbs, 1, null, true);
    return [logits, actions];
  });
}
```

Преобразование логитов в значения вероятности приложения силы влево

Вычисление значений вероятности для обоих действий, необходимое для функции tf.multinomial()

Случайная выборка действий, исходя из значений вероятности. Четыре аргумента представляют собой значения вероятности, количество примеров данных, случайное начальное значение (не используется) и флаг, указывающий, что значения вероятности нормализованы

11.2.3. Обучение сети стратегий: алгоритм REINFORCE

Основной вопрос теперь: как вычислить, какие действия удачные, а какие — нет. После ответа на него можно обновить весовые коэффициенты сети стратегий так, чтобы повысить вероятность выбора удачных действий в будущем, подобно обучению с учителем. На ум сразу приходит возможность оценки удачности действий на основе вознаграждений. Но вознаграждения в задаче удержания равновесия шеста на тележке отличаются: 1) фиксированным значением (1) и 2) получением агентом вознаграждения на каждом шаге, вплоть до завершения эпизода. Поэтому нельзя просто воспользоваться пошаговым вознаграждением в качестве метрики, иначе все действия будут считаться одинаково удачными. Необходимо учитывать длительность эпизодов.

«Наивный» подход состоит в суммировании всех вознаграждений эпизода и выяснении таким образом его длительности. Но хорошо ли такая сумма позволяет оценить действия? Несложно понять, что она здесь не подходит. Причина в ходах в конце эпизода. Представьте себе длительный эпизод, когда агент прекрасно балансирует системой «шест — тележка» почти до самого конца, затем выбирает несколько неудачных вариантов, в результате чего эпизод завершается. При «наивном» подходе с суммированием у неудачных действий в конце эпизода и удачных предшествующих действий окажутся одинаково хорошие оценки. Нам же хотелось бы присвоить более высокие оценки действиям в начале и середине эпизода и более низкие — действиям в конце эпизода.

Это приводит нас к простой, но играющей важную в RL роль идее *дисконтирования вознаграждений* (reward discounting): величина для конкретного хода должна равняться немедленному вознаграждению плюс ожидаемое в будущем вознагра-

ждение. Будущее вознаграждение может играть столь же важную роль, что и немедленное, но его важность может быть и меньше. Выразить количественно это относительное равновесие можно с помощью коэффициента дисконтирования γ (гамма). Его значение обычно близко к 1, но чуть меньше, например 0,95 или 0,99. Это можно выразить следующим математическим уравнением:

$$v_i = r_i + \gamma \cdot r_{i+1} + \gamma^2 \cdot r_{i+2} + ... + \gamma^{N-i} r_N. \quad (11.1)$$

В уравнении (11.1) v_i — общее дисконтированное вознаграждение состояния на шаге i, которое можно считать ценностью конкретного состояния. Оно равно немедленному вознаграждению агента на этом шаге (r_i) плюс вознаграждение со следующего шага (r_{i+1}), все это дисконтировано на коэффициент γ, плюс дисконтированное вознаграждение через два шага и так далее, вплоть до конца эпизода (шаг N).

Чтобы проиллюстрировать дисконтирование вознаграждения, мы покажем, как это уравнение преобразует исходные вознаграждения в более удобную метрику ценности (рис. 11.5). Верхний график в блоке А демонстрирует исходные вознаграждения с четырех шагов короткого эпизода. На нижнем графике показаны дисконтированные вознаграждения (согласно уравнению (11.1)). В блоке Б для сравнения приведены исходные и дисконтированные вознаграждения с более длительного эпизода (длина — 20). В этих двух блоках видно, что величина суммарного дисконтированного вознаграждения выше в начале и ниже в конце эпизода, что логично, ведь мы как раз и хотим поставить более низкие значения в соответствие действиям в конце эпизода, приводящим к завершению игры. Кроме того, ценность в начале и середине более длительного эпизода (блок Б) выше, чем ценность в начале более короткого (блок А). Это также интуитивно понятно, поскольку мы хотим поставить более высокую ценность в соответствие действиям, приводящим к удлинению эпизода.

В результате использования уравнения дисконтирования вознаграждения получается набор более логичных, чем при «наивном» подходе, значений. Но все равно остается вопрос: как обучить сеть стратегий с помощью этих дисконтированных величин вознаграждений? Для этого мы воспользуемся алгоритмом REINFORCE, изобретенным Рональдом Вильямсом в 1992 году[1]. Основная идея алгоритма — подобрать значения весовых коэффициентов сети стратегий так, чтобы повысить вероятность выбора ею удачных вариантов (вариантов, которым присвоены большие величины дисконтированного вознаграждения) и снизить вероятность выбора вариантов неудачных (которым присвоены меньшие величины дисконтированного вознаграждения).

Для этого нам необходимо вычислить, в каком направлении следует менять параметры, чтобы повысить вероятность действия, при данных входных наблюдениях. Это реализует код из листинга 11.3 (фрагмент из `cart-pole/index.js`). На каждом ходе игры вызывается функция `getGradientsAndSaveActions()`, сравнивающая логиты (ненормализованные оценки вероятности) с фактически выбираемым на этом ходе действием и возвращающая градиент этого расхождения относительно весов

[1] *Williams R. J.* Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning // Machine Learning. Vol. 8. Nos. 3–4. Pp. 229–256. http://mng.bz/WOyw.

сети стратегий. Хотя этот алгоритм и может показаться сложным, интуитивно он очень прост. Возвращаемый градиент указывает сети стратегий, как нужно изменить весовые коэффициенты, чтобы выбирать варианты, более похожие на фактически выбранные. Градиенты в совокупности с вознаграждениями от эпизодов обучения и составляют основу данного метода обучения с подкреплением. Именно поэтому он относится к семейству методов RL, которые называются *градиентным спуском по стратегиям* (policy gradients).

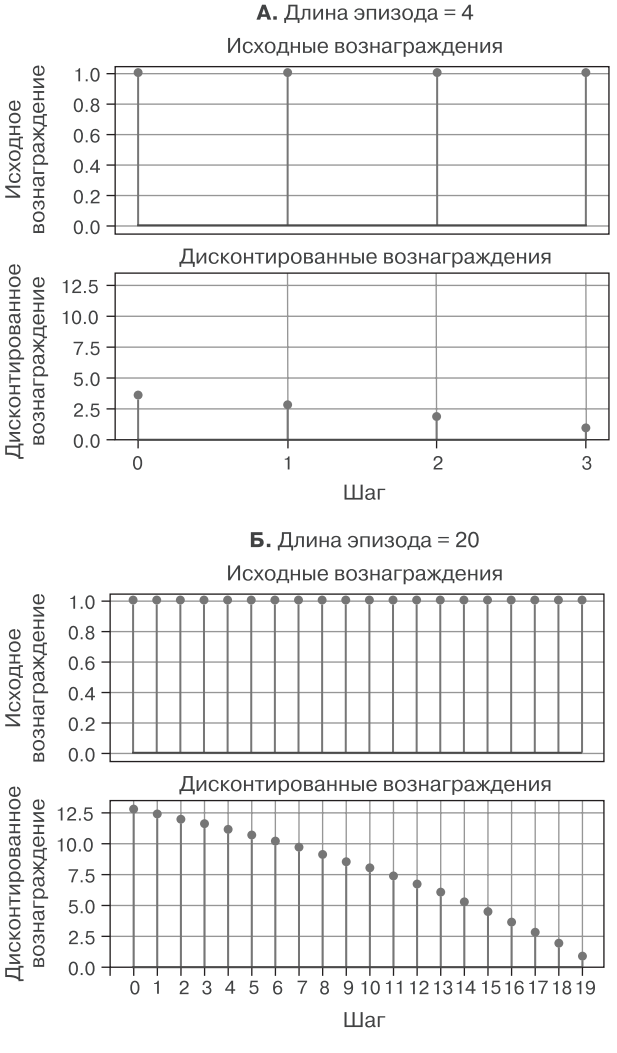


Рис. 11.5. Блок А: применение дисконтирования (уравнение (11.1)) к вознаграждениям эпизода длительностью четыре хода. Блок Б: то же самое, что и в блоке А, только длительность эпизода составляет 20 ходов (то есть в пять раз больше, чем в блоке А). В результате дисконтирования действия в начале каждого эпизода обладают большей ценностью, чем действия ближе к его концу

Листинг 11.3. Получаем градиенты для весовых коэффициентов, сравнивая логиты и фактические действия

```
getGradientsAndSaveActions(inputTensor) {
  const f = () => tf.tidy(() => {
    const [logits, actions] =
        this.getLogitsAndActions(inputTensor);
    this.currentActions_ = actions.dataSync();
    const labels =
        tf.sub(1, tf.tensor2d(this.currentActions_, actions.shape));
    return tf.losses.sigmoidCrossEntropy(
        labels, logits).asScalar();
  });
  return tf.variableGrads(f);
}
```

Функция getLogitsAndActions() описана в листинге 11.2

Потери на основе перекрестной энтропии с сигма-функцией активации численно выражают расхождение между фактическим действием, произведенным во время игры и выходными логитами сети стратегий

Вычисляет градиент функции потерь относительно весовых коэффициентов сети стратегий

Во время обучения агент получает возможность сыграть некоторое количество игр (скажем, N) и собрать все дисконтированные вознаграждения в соответствии с уравнением 11.1, а также градиенты со всех ходов. Далее мы вычисляем сочетание дисконтированных вознаграждений с градиентами, умножая градиенты на нормализованную версию дисконтированных вознаграждений. Нормализация вознаграждений играет здесь важную роль, смещая линейно и масштабируя все дисконтированные вознаграждения от N игр так, чтобы их суммарное среднее значение было равно 0, а суммарное среднеквадратичное отклонение — 1. Пример применения такой нормализации к дисконтированным вознаграждениям приведен на рис. 11.6, где показаны нормализованные дисконтированные вознаграждения в случае краткого (длина — 4) и более длинного (длина — 20) эпизода. Из этого рисунка должно быть ясно, каким шагам отдает предпочтение алгоритм REINFORCE: действиям, произведенным в начале и середине более длинного эпизода. И напротив, всем шагам более короткого эпизода (длина — 4) присваиваются *отрицательные* величины. Что означает отрицательное нормализованное вознаграждение? А то, что при обновлении на его основе весовых коэффициентов сети стратегий в будущем сеть при получении такого же состояния на входе будет стремиться выбрать *другой* курс действий. В отличие от положительного вознаграждения, при котором сеть будет стремиться в будущем выбирать *такие же* действия при получении такого же состояния на входе.

Код для нормализации дисконтированных вознаграждений и масштабирование с его помощью градиентов довольно громоздки, но не сложны. Ради экономии места мы не станем его здесь приводить, но найти его можно в функции `scaleAndAverageGradients()` файла `cart-pole/index.js`. Для обновления весовых коэффициентов сети стратегий используются масштабированные градиенты. После обновления весовых коэффициентов сеть стратегий выдает на выходе бóльшие логиты для действий с тех шагов, которым были поставлены в соответствие более высокие дисконтированные вознаграждения, и меньшие логиты — для действий с тех шагов, которым были поставлены в соответствие более низкие дисконтированные вознаграждения.

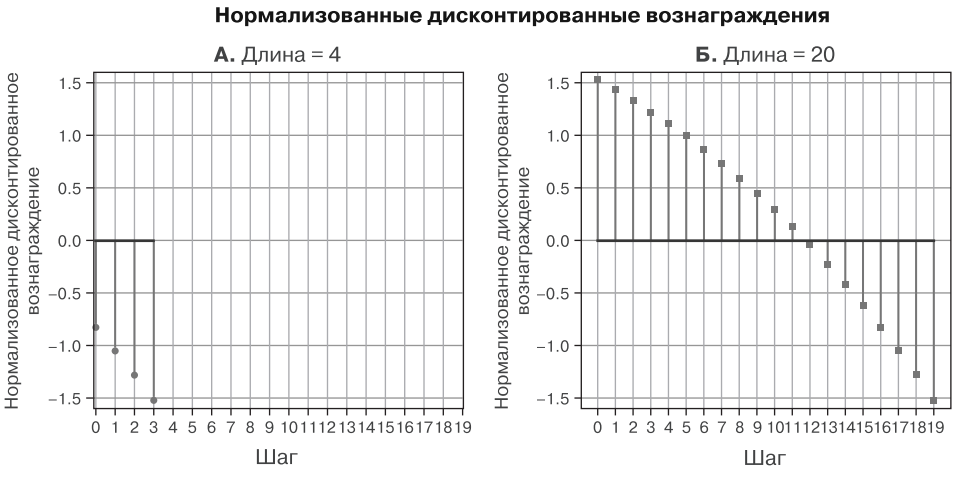


Рис. 11.6. Нормализация дисконтированных вознаграждений для двух эпизодов: длиной 4 (блок А) и длины 20 (блок Б). Как видим, максимальная величина у нормализованных дисконтированных вознаграждений — в начале эпизода длиной 20. Метод градиентного спуска по стратегиям обновит на основе этих величин дисконтированных вознаграждений весовые коэффициенты сети стратегий так, что она с меньшей вероятностью выберет в будущем те действия, которые привели к низким вознаграждениям в первом случае (длина — 4) и с большей вероятностью — действия, приведшие к высоким вознаграждениям в начальной части второго эпизода (длина — 20) (при тех же входных состояниях, конечно)

Фактически так и работает алгоритм REINFORCE. Базовая логика обучения примера cart-pole, в основе которого лежит алгоритм REINFORCE, приведена в листинге 11.4. Она представляет собой многократное повторение описанных ниже шагов.

1. Вызов сети стратегий и получение логитов на основе текущих наблюдений агента.
2. Случайная выборка действия на основе логитов.
3. Обновление среды на основе выбранного действия.
4. Запоминание такой информации для дальнейшего обновления весовых коэффициентов (на шаге 7), как: логиты и выбранное действие, а также градиенты функции потерь относительно весов сети стратегий. Эти градиенты называются *градиентами по стратегиям* (policy gradients).
5. Получение вознаграждения от среды и запоминание его на будущее (шаг 7).
6. Повторение шагов 1–5 до тех пор, пока не будет завершено `numGames` эпизодов.
7. По завершении `numGames` эпизодов выполнение дисконтирования и нормализации вознаграждений и масштабирование градиентов из шага 4 на основе полученных результатов. Далее обновление весовых коэффициентов сети стратегий на основе масштабированных градиентов (именно в этом месте обновляются весовые коэффициенты сети стратегий).
8. (Не показано в листинге 11.4.) Повторение шагов 1–7 `numIterations` раз.

Сравните эти шаги с кодом из листинга (фрагмент `cart-pole/index.js`), чтобы проследить соответствия и убедиться, что вы понимаете логику.

Листинг 11.4. Цикл обучения примера cart-pole, реализующий алгоритм REINFORCE

```
async train(
    cartPoleSystem, optimizer, discountRate, numGames, maxStepsPerGame) {
  const allGradients = [];
  const allRewards = [];
  const gameSteps = [];
  onGameEnd(0, numGames);
  for (let i = 0; i < numGames; ++i) {
    cartPoleSystem.setRandomState();
    const gameRewards = [];
    const gameGradients = [];
    for (let j = 0; j < maxStepsPerGame; ++j) {
      const gradients = tf.tidy(() => {
      const inputTensor = cartPoleSystem.getStateTensor();
        return this.getGradientsAndSaveActions(
            inputTensor).grads;
      });

      this.pushGradients(gameGradients, gradients);
      const action = this.currentActions_[0];
      const isDone = cartPoleSystem.update(action);
      await maybeRenderDuringTraining(cartPoleSystem);

      if (isDone) {
        gameRewards.push(0);
        break;
      } else {
        gameRewards.push(1);
      }
    }
    onGameEnd(i + 1, numGames);
    gameSteps.push(gameRewards.length);
    this.pushGradients(allGradients, gameGradients);
    allRewards.push(gameRewards);
    await tf.nextFrame();
  }

  tf.tidy(() => {
    const normalizedRewards =
        discountAndNormalizeRewards(allRewards, discountRate);
    optimizer.applyGradients(
        scaleAndAverageGradients(allGradients, normalizedRewards));
  });
  tf.dispose(allGradients);
  return gameSteps;
}
```

Проходим в цикле заданное количество эпизодов

Задание случайных начальных значений эпизода игры

Проходим в цикле по ходам игры

Отслеживаем градиенты со всех ходов для дальнейшего обучения с помощью алгоритма REINFORCE

Агент выполняет в среде действие

Агент получает единичное вознаграждение при каждом ходе, вплоть до завершения игры

Дисконтирование и нормализация вознаграждений (ключевой шаг алгоритма REINFORCE)

Обновление весовых коэффициентов сети стратегий на основе масштабированных градиентов со всех шагов

Чтобы увидеть алгоритм REINFORCE в действии, задайте 25 эпох на странице демонстрации и нажмите кнопку Train. По умолчанию состояние среды отображается в режиме реального времени во время обучения, так что можно видеть, как обучающийся агент делает многократные пробы. Для ускорения обучения снимите флажок Render During Training (Отрисовка во время обучения). Двадцать пять эпох обучения займет несколько минут на более или менее современном ноутбуке. Этого должно

быть достаточно для достижения наилучших показателей работы (при настройках по умолчанию: 500 ходов на каждый эпизод игры). Рисунок 11.7 демонстрирует типичную кривую обучения, на нем построен график средней длины эпизода как функции от номера итерации обучения. Обратите внимание, что в процессе обучения наблюдаются довольно сильные колебания, среднее число ходов меняется с итерациями немонотонным и сильно зашумленным образом. Для обучения с подкреплением подобные колебания характерны.

По завершении обучения нажмите кнопку Test и увидите, что агент весьма неплохо удерживает систему «тележка — шест» в равновесии на протяжении значительного числа ходов. А поскольку на этапе контроля нет ограничений максимального числа ходов (500 по умолчанию), возможно, агент сможет растянуть эпизод и на более чем 1000 ходов. Если эпизод длится слишком долго, можете нажать кнопку Stop и моделирование завершится.

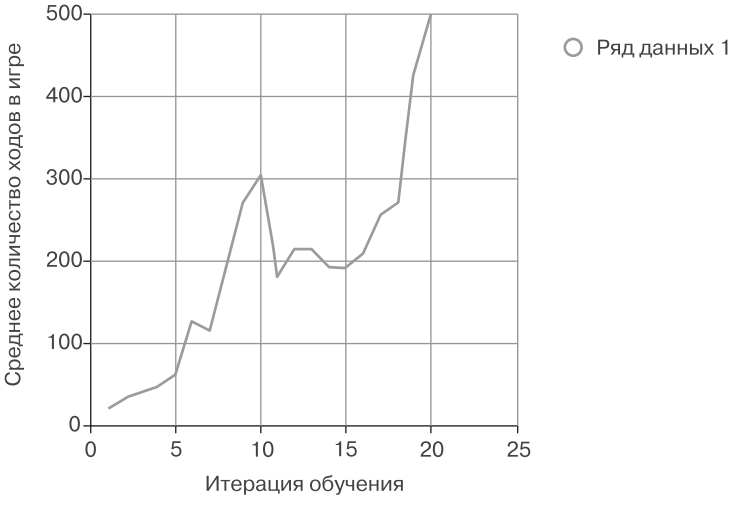


Рис. 11.7. Кривая, отражающая количество ходов, которое агент способен продержаться в эпизодах примера cart-pole, как функция от количества итераций обучения. Идеальный результат (в данном случае 500 шагов) достигается примерно на 20-й итерации. Этот результат достигается при размере скрытого слоя, равном 128 нейронам. Заметно немонотонная и колеблющаяся форма кривой довольно характерна для задач RL

В завершение этого раздела подытожим постановку задачи и роль алгоритма REINFORCE градиентов по стратегиям на рис. 11.8. На этом рисунке изображены основные части решения. На каждом шаге агент с помощью нейронной *сети стратегий* оценивает степень правдоподобия того, что приложение направленной влево (или, что эквивалентно, вправо) силы — более удачный вариант. Эта степень правдоподобия преобразуется в фактическое действие с помощью процесса случайной выборки, побуждающего агент исследовать на ранних этапах и действовать в соответствии со степенью достоверности позднее. Данное действие меняет состояние системы «тележка — шест» в среде, которая, в свою очередь, вознаграждает агент вплоть до конца эпизода. Процесс повторяется на протяжении нескольких эпизо-

дов, во время чего алгоритм REINFORCE запоминает вознаграждение, действие и возвращаемую сетью стратегий оценку на каждом шаге. Перед обновлением сети стратегий алгоритм REINFORCE различает полученные от сети хорошие и плохие оценки, путем дисконтирования и нормализации, и на основе полученных результатов направляет весовые коэффициенты сети в стороны улучшения оценок в будущем. Процесс повторяется несколько раз, вплоть до конца обучения (например, достижения агентом заданного порогового значения качества работы).

Отвлечемся на минуту от всех технических подробностей и взглянем на общую картину воплощенного в этом примере обучения с подкреплением. Подход на основе RL явно превосходит методы, не связанные с машинным обучением, например классическую теорию управления, своей универсальностью и экономией человеческих усилий. Для сложных систем или систем, характеристики которых неизвестны, RL может оказаться единственным работоспособным подходом. А если характеристики системы меняются с течением времени, можно не выводить новые математические решения с нуля, а просто заново запустить алгоритм RL и позволить агенту приспосабливаться к новой ситуации.

Недостаток подхода RL, до сих пор остающийся нерешенной проблемой в сфере исследований обучения с подкреплением, состоит в том, что в среде необходимо выполнять множество повторяющихся проб. В случае примера с шестом и тележкой достижение целевого уровня навыков агента требует около 400 эпизодов игры. В некоторых традиционных, не RL-подходах таких проб вообще не нужно. Достаточно реализовать алгоритм на основе теории управления и агент сможет удерживать шест в равновесии с первого же эпизода. Для задач а-ля «шест — тележка» стремление RL к повторению проб не большая проблема в силу простоты, быстроты и малого количества ресурсов, необходимых для компьютерного моделирования подобной среды. Однако в приближенных к реальности задачах, например беспилотных автомобилях и механических руках-манипуляторах, эта проблема RL встает весьма остро. Никто не может позволить себе сотни или тысячи раз разбивать автомобиль или ломать руку робота для обучения агента, не говоря уже об огромном количестве времени, которое займет выполнение алгоритма обучения с подкреплением для подобных реальных задач.

На этом наш первый пример RL завершается. У задачи тележки с шестом есть особенности, не свойственные другим задачам RL. Например, многие среды RL не вознаграждают положительно агент на каждом шаге. В некоторых случаях ему приходить принимать десятки, если не больше, решений для получения положительного вознаграждения. В промежутке между положительными вознаграждениями он может вообще не получать никакого вознаграждения либо получать только отрицательное (хотя, кажется, таковы многие начинания в реальном мире, например учеба, тренировки и инвестиции!). Кроме того, у системы тележка — шест отсутствует «память» в том смысле, что динамика системы не зависит от предыдущих действий агента. Большинство задач RL более сложны и действия агента меняют определенные аспекты среды. В следующем разделе мы рассмотрим задачу RL, отличающуюся как разреженными положительными вознаграждениями, так и изменениями среды в результате действий. Для решения этой задачи мы познакомим вас еще с одним популярным полезным алгоритмом RL — *глубоким Q-обучением* (deep Q-learning).

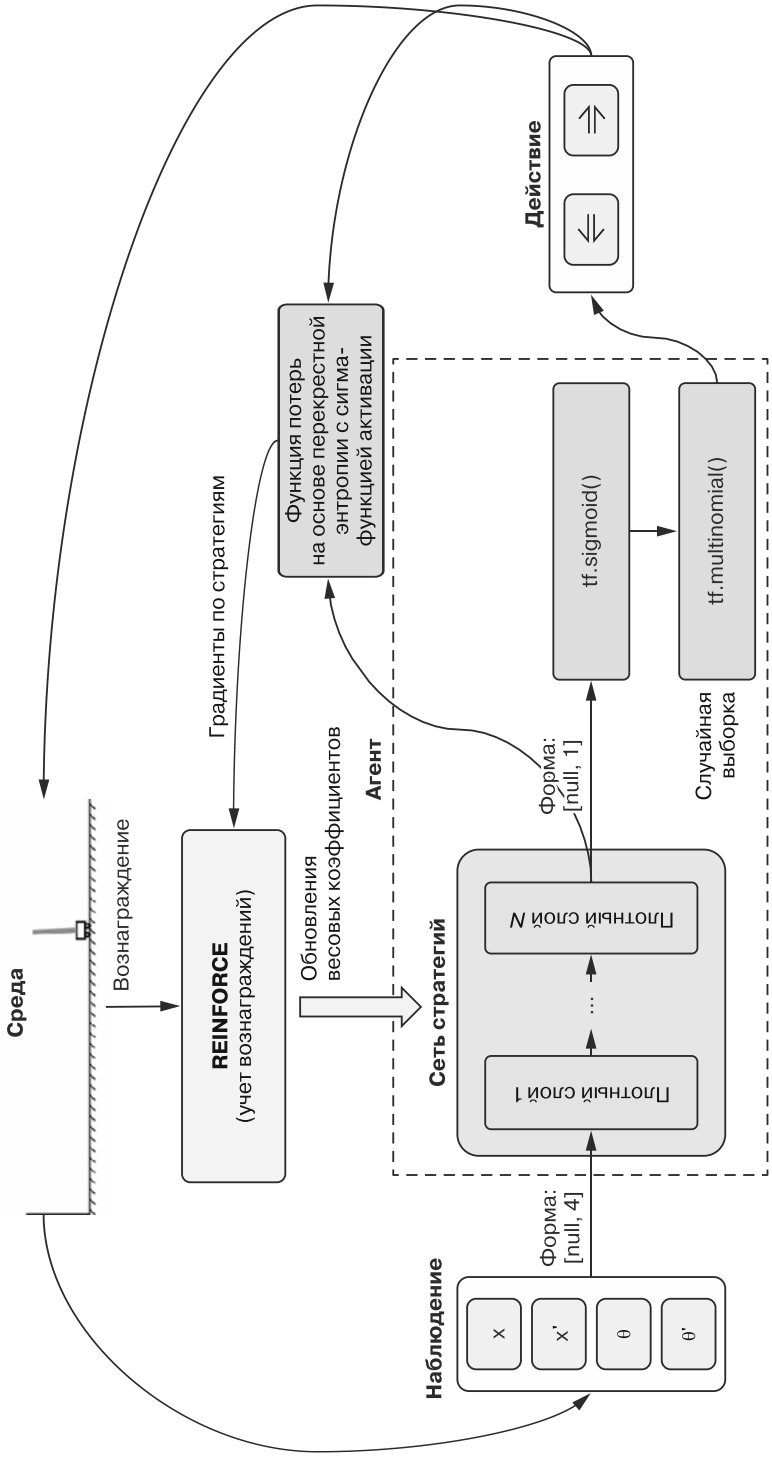


Рис. 11.8. Схематическая иллюстрация основанного на алгоритме REINFORCE подхода к решению задачи по удержанию равновесия шеста на тележке. Эта схема — расширенный вариант схемы с рис. 11.4

11.3. Оценочные сети и Q-обучение: пример игры «Змейка»

В качестве примера для обсуждения Q-обучения возьмем классическую экшен-игру «Змейка». Как и в предыдущем разделе, мы сначала сформулируем задачу RL и проблемы, которые она ставит перед разработчиками. В ходе этого мы также обсудим, почему градиенты по стратегиям и алгоритм REINFORCE плохо подходят для ее решения.

11.3.1. «Змейка» как задача обучения с подкреплением

Впервые появившись среди аркад 1970-х годов, «Змейка» быстро стала популярным жанром компьютерных игр. Каталог snake-dqn из репозитория tfjs-examples включает JavaScript-реализацию простого ее варианта. Извлечь код можно с помощью следующих команд:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/snake-dqn
yarn
yarn watch
```

На открываемой командой `yarn watch` веб-странице вы увидите доску для игры «Змейка». Можете загрузить размещенную на сервере предобученную модель глубокой Q-сети (deep Q-network, DQN) и посмотреть, как она играет в игру. Позднее мы поговорим о том, как обучить подобную модель с нуля, а пока что вам достаточно понаблюдать за игрой, чтобы составить себе представление о том, как она работает. На случай, если игра «Змейка» вам незнакома, вот ее основные правила и настройки.

Во-первых, все действия происходят в мире размером 9×9 (см. пример на рис. 11.9). Этот мир (доску) можно увеличить, но в нашем примере по умолчанию размер равен 9×9. Клетки доски делятся на три типа: змейка, фрукты и пустое пространство. Змейка состоит из синих клеток, за исключением головы, окрашенной в оранжевый цвет с полукругом, изображающим ее рот. Фрукты изображаются в виде зеленых клеток с кругом внутри. Пустые клетки — белого цвета. Игра состоит из ходов — или, на языке компьютерных игр, *кадров* (frame). На каждом ходе агент должен выбрать одно из трех возможных действий змейки: двигаться прямо, повернуть налево или повернуть направо (оставаться на месте нельзя). Агент получает положительное вознаграждение, если голова змейки соприкасается с клеткой фрукта, в случае чего клетка фрукта исчезает (змейка ее «съедает»), длина змейки увеличивается на единицу со стороны хвоста, а на одной из пустых клеток появляется новый фрукт. Игра завершается (змейка «умирает»), когда голова змейки выходит за границы (как в блоке Б на рис. 11.9) или натыкается на ее собственное тело (как в блоке В).

Одна из главных трудностей при игре в «Змейку» — ее (змейки) рост. Если бы не это правило, игра была бы намного проще и можно было бы просто направлять змейку на фрукты все снова и снова, а вознаграждение агента было бы потенциально неограниченным. При наличии правила о росте змейки, однако, агент должен

научиться не натыкаться на свое тело, что усложняется по мере того, как змейка поедает фрукты и становится все длиннее. Этот аспект делает задачу обучения с подкреплением змейки нестатической, в отличие от среды «тележка — шест», как мы упоминали в конце предыдущего раздела.

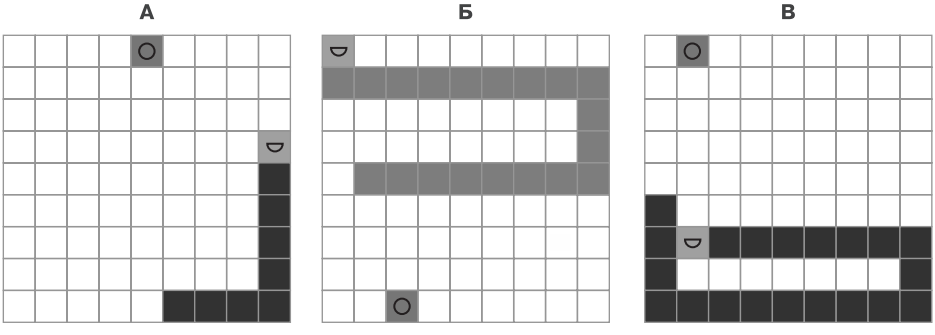


Рис. 11.9. Игра «Змейка»: доска, на которой игрок управляет змейкой, поедающей фрукты. «Цель» змейки — съесть как можно больше фруктов благодаря оптимальной траектории движения. С каждым съеденным фруктом длина змейки увеличивается на одну клетку. Игра завершается (змейка «умирает»), как только змейка выходит за границы (блок Б) или натыкается на свое собственное тело (блок В). Обратите внимание, как в блоке Б голова змейки достигает границы, после чего движется вперед (прямо), в результате чего игра завершается. Если змейка просто достигает клеток на границе, игра сразу не завершается. Поедание всех фруктов приводит к большому положительному вознаграждению. Перемещение на одну клетку без поедания фрукта приводит к меньшему по модулю отрицательному вознаграждению. Завершение игры («гибель» змейки) также вызывает отрицательное вознаграждение

Таблица 11.2 описывает задачу змейки в канонической постановке задач RL.

Таблица 11.2. Описание задачи змейки в канонической постановке RL

Абстрактное понятие RL	**Воплощение в задаче змейки**
Среда	Игровая доска, включающая движущуюся змейку и пополняющийся сам собой запас фруктов
Действие	(Дискретное.) Выбор из трех действий: двигаться прямо, повернуть налево или повернуть направо
Вознаграждение	(Частое, смешанно положительно-отрицательное): • поедание фрукта приносит большое положительное вознаграждение (+10); • перемещение без поедания фрукта приносит небольшое отрицательное вознаграждение (–0,2); • гибель змейки — большое отрицательное вознаграждение (–10)
Наблюдение	(Полное состояние, дискретное.) На каждом ходе агент обладает доступом к полному состоянию системы: видит состояние всех клеток доски

Основное отличие, по сравнению с постановкой задачи «тележка — шест» (см. табл. 11.1), состоит в структуре вознаграждения. В задаче змейки агент получает положительные вознаграждения (+10 за каждый съеденный фрукт) лишь изредка — то есть лишь после определенного количества отрицательных вознаграждений, вызванных перемещениями змейки, необходимыми для достижения фрукта. При данном размере доски два положительных вознаграждения могут отстоять друг от друга на 17 ходов, даже если змейка движется самым оптимальным образом. Небольшое отрицательное вознаграждение — мера, заставляющая змейку двигаться по самому короткому пути. Без этой меры змейка могла бы петлять самым замысловатым путем и все равно получить то же вознаграждение, что излишне затянуло бы игру и процесс обучения. Кроме того, вследствие именно подобной разреженной сложной структуры вознаграждений главным образом и не получится воспользоваться для решения этой задачи градиентным спуском по стратегиям и методом REINFORCE.

JavaScript-API змейки

Нашу реализацию игры «Змейка» вы можете найти в файле `snake-dqn/snake_game.js`. Мы опишем только API класса `SnakeGame` и избавим вас от утомительных подробностей реализации, которые вы можете при желании изучить сами. Синтаксис конструктора класса `SnakeGame` — следующий:

```
const game = new SnakeGame({height, width, numFruits, initLen});
```

Размеры доски по умолчанию, `height` и `width`, равны 9. `numFruits` — количество присутствующих на доске в любой момент времени фруктов; по умолчанию — 1. `initLen` — начальная длина змейки — по умолчанию равна 2.

Метод `step()` объекта `game` позволяет вызывающей стороне сделать ход в игре:

```
const {state, reward, done, fruitEaten} = game.step(action);
```

Аргумент метода `step()` отражает выполняемое действие: 0 — двигаться прямо, 1 — повернуть налево, 2 — повернуть направо. Возвращаемое методом `step()` значение включает следующие поля.

- `state` — новое состояние доски, сразу же после действия, в виде простого объекта JavaScript с двумя полями:
  - `s` — занимаемые змейкой клетки в виде массива координат `[x, y]`. Элементы массива упорядочены: первый элемент соответствует голове змейки, а последний — хвосту;
  - `f` — координаты `[x, y]` занимаемых фруктами клеток.

  Отметим, что это представление состояния игры специально сделано как можно более экономичным, поскольку алгоритму Q-обучения приходится хранить большое число (например, десятки тысяч) подобных объектов состояния, как мы вскоре увидим. Или же можно использовать массив или вложенный массив для фиксации состояния каждой клетки доски, включая пустые. Но при этом память будет использоваться намного менее эффективно.

- `reward` — вознаграждение, получаемое змейкой при каждом ходе сразу же после действия. Представляет собой одно число.
- `done` — булев флаг, указывающий, завершается ли игра сразу же после данного действия.
- `fruitEaten` — булев флаг, указывающий, был ли съеден на данном ходе змейкой фрукт в результате данного действия. Учтите, что это поле избыточно, поскольку вычислить, был ли съеден фрукт, можно на основе поля `reward`. Оно включено для упрощения, а также расцепления конкретных величин вознаграждений (которые могут играть роль настраиваемых гиперпараметров) с бинарным событием: съеден фрукт или не съеден.

Как мы увидим далее, первые три поля (`state`, `reward` и `done`) играют важную роль в алгоритме Q-обучения, в то время как последнее поле (`fruitEaten`) служит в основном для мониторинга.

11.3.2. Марковский процесс принятия решений и Q-значения

Для описания алгоритма глубокого Q-обучения, которым мы воспользуемся для задачи змейки, нам придется немного углубиться в математику. В частности, мы познакомим вас с *марковским процессом принятия решений* (Markov decision process, MDP) и математическим аппаратом, лежащим в его основе. Не волнуйтесь: мы приведем простые и конкретные примеры и привяжем излагаемые понятия к задаче змейки.

С точки зрения MDP история среды RL представляет собой последовательность переходов между конечным множеством дискретных состояний. Кроме того, эти переходы между состояниями удовлетворяют определенному правилу: *«Состояние среды на следующем шаге определяется исключительно текущим состоянием и предпринимаемым агентом на текущем шаге действием»*.

Ключевой момент: следующее состояние зависит *только* от текущего состояния и предпринимаемого действия и более ни от чего. Другими словами, MDP предполагает, что история процесса (каким образом вы попали в текущее состояние) не должна играть никакой роли при определении, что делать далее. Это колоссальное упрощение задачи. А что представляет собой *не марковский процесс принятия решений*? Случай, когда следующее состояние зависит не только от текущих состояния и действия, но и от состояний или действий на предыдущих шагах, возможно начиная с самого начала эпизода. Математика не марковского сценария значительно сложнее, и для вычислений понадобится намного больше вычислительных ресурсов.

Удаление требованиям MDP для многих задач RL интуитивно понятны. Хороший пример — шахматы. На любом ходе игры позиция на доске (плюс то, чей ход сейчас) полностью характеризует состояние игры и дает всю информацию, необходимую игроку для вычисления следующего хода. Другими словами, игрок может продолжить игру с данной позиции, не зная предыдущих ходов (кстати, именно поэтому газеты могут печатать шахматные задачи, расходуя очень мало места).

Компьютерные игры наподобие змейки также попадают под формулировку MDP. Расположение на доске змейки и фруктов полностью характеризует состояние игры, этой информации вполне достаточно для продолжения игры с текущего момента или для выбора агентом следующего действия.

И хотя игры наподобие шахмат и «Змейки» прекрасно согласуются с требованиями MDP, число состояний в них зачастую чрезвычайно велико. Для интуитивно понятной и наглядной иллюстрации MDP нам понадобится более простой пример. На рис. 11.10 мы покажем очень простую задачу MDP, в которой есть всего семь возможных состояний и два возможных действия агента. Переходы между состояниями определяются следующими правилами.

- Начальное состояние — всегда s_1.
- Из состояния s_1 при выполнении агентом действия a_1 среда переходит в состояние s_2. При выполнении же агентом действия a_2 среда переходит в состояние s_3.
- Из каждого из состояний s_2 и s_3 переход среды в следующее состояние определяется аналогичным набором правил ветвления.
- Состояния s_4, s_5, s_6 и s_7 — завершающие: по достижении любого из этих состояний эпизод завершается.

Итак, каждый из эпизодов этой задачи RL длится ровно три шага. Каким образом агент в этой задаче RL выбирает действие на первом и втором шагах? Поскольку речь идет о задаче RL, данный вопрос имеет смысл только в терминах вознаграждений. В MDP каждое действие приводит не только к переходу в другое состояние, но и получению вознаграждения. На рис. 11.10 вознаграждения изображены в виде стрелок, соединяющих действия со следующими состояниями и помеченных `r = <величина_вознаграждения>`. Цель агента, конечно, — максимизировать суммарное вознаграждение (дисконтированное на определенный коэффициент). Теперь представьте себе, что вы агент на первом шаге. Давайте задумаемся, какие рассуждения позволят нам решить, какое действие — a_1 или a_2 — лучше. Пусть коэффициент дисконтирования вознаграждения (γ) равен 0,9.

Ход рассуждений может быть следующим. Если выбрать действие a_1, мы немедленно получим вознаграждение –3 и перейдем в состояние s_2. Если выбрать действие a_2, мы немедленно получим вознаграждение 3 и перейдем в состояние s_3. Значит ли это, что a_2 — лучший вариант, поскольку 3 больше, чем –3? Ответ: нет, поскольку 3 и –3 — всего лишь немедленные вознаграждения и мы не учли вознаграждения с последующих шагов. Необходимо найти *наилучший возможный* исход для s_2 и s_3. Какой исход для s_2 — наилучший? Ответ: порождаемый действием a_2, которое дает вознаграждение 10. Таким образом, максимальное дисконтированное вознаграждение, которое только можно ожидать, если выполнить в состоянии s_1 действие a_1, равно:

Максимальное вознаграждение при действии a_1 в состоянии s_1	= немедленное вознаграждение + дисконтированное будущее вознаграждение = $= -3 + \gamma \cdot 10 =$ $= -3 + 0{,}9 \cdot 10 =$ $= 6$

Аналогично наилучший исход для состояния s_3 — при действии a_1, что дает вознаграждение –4. Следовательно, если выполнить в состоянии s_1 действие a_2, максимальное дисконтированное вознаграждение будет:

Максимальное вознаграждение при действии a_2 из состояния s_1	= немедленное вознаграждение + дисконтированное будущее вознаграждение = $= 3 + \gamma \cdot -4 =$ $= 3 + 0{,}9 \cdot -4 =$ $= 0{,}6$

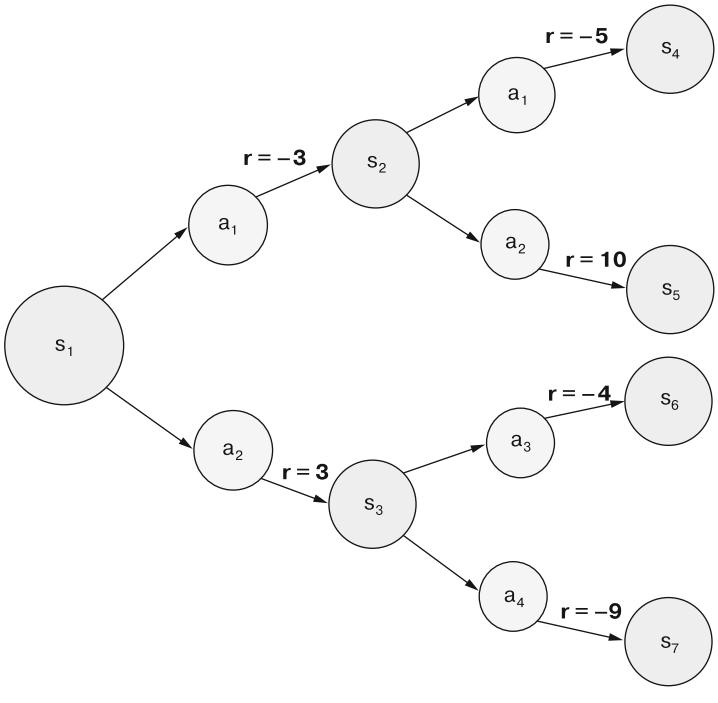


Рис. 11.10. Очень простой конкретный пример марковского процесса принятия решений (MDP). Состояния изображены в виде серых кругов с метками s_n, а действия — серых кругов с метками a_m. Вознаграждения, соответствующие вызванным действиями переходам из состояния в состояние, обозначены $r = x$

Вычисленные нами дисконтированные вознаграждения — примеры так называемых *Q-значений* (Q-values). Q-значение — это ожидаемое совокупное вознаграждение (с учетом дисконтирования) для действия в заданном состоянии. Из этих Q-значений ясно, что в состоянии s_1 действие a_1 лучше, чем a_2, — умозаключение, отличное от того, к которому мы пришли на основе величин одних немедленных вознаграждений при первом действии. В упражнении 3 в конце главы у вас будет возможность поупражняться в вычислении Q-значений в более реалистичных сценариях MDP, в которых присутствует стохастичность.

Приведенный пример хода рассуждений может показаться тривиальным. Но из него следует абстракция, играющая ключевую роль в Q-обучении. Q-значение, обозначаемое $Q(s, a)$, представляет собой функцию текущего состояния (s) и действия (a). Другими словами, функция $Q(s, a)$ ставит паре «состояние — действие» в соответствие оценку величины вознаграждения от выполнения данного действия в данном состоянии. Эта величина лишь перспективная, поскольку учитывает максимально возможные будущие вознаграждения, в допущении, что на всех будущих шагах будут производиться оптимальные действия.

Благодаря этому для выбора оптимального действия в любом конкретном состоянии нам достаточно величины $Q(s, a)$. В частности, при известной $Q(s, a)$ оптимальным будет действие с максимальным Q-значением среди всех возможных действий:

$$\text{значение } a, \text{ при котором достигается максимум из } Q(s_i, s_1), Q(s_i, a_2) \ldots Q(s_i, a_N), \tag{11.2}$$

где N — число всех возможных действий. При наличии хорошей оценки $Q(s, a)$ можно просто следовать на каждом шаге этому процессу принятия решений и гарантированно получить максимально возможное совокупное вознаграждение. Таким образом, задача RL поиска оптимального процесса принятия решений сводится к поиску путем обучения функции $Q(s, a)$. Отсюда и название алгоритма обучения: Q-обучение.

Давайте на минуту отвлечемся и посмотрим, чем Q-обучение отличается от метода градиентного спуска по стратегиям, который мы использовали в задаче удержания шеста на тележке в равновесии. В градиентном спуске по стратегиям идея заключалась в предсказании наилучшего действия из возможных; а в Q-обучении — в предсказании значений ценности для всех возможных действий. И если градиенты по стратегиям непосредственно указывают, какое действие следует выбрать, Q-обучение идет слегка окольным путем и требует дополнительного шага «выбрать максимальное значение». Этот окольный путь позволяет проще установить связь между вознаграждениями и значениями ценности для последующих шагов, облегчая тем самым обучение в задачах с разреженными положительными вознаграждениями, наподобие задачи змейки.

Так какова же связь между вознаграждениями и значениями ценности для последующих шагов? Мы уже видели эту связь мельком, когда решали простую задачу MDP на рис. 11.10. Математически ее можно выразить следующим образом:

$$Q(s_i, a) = r + \gamma \cdot [\text{максимальное значение из } Q(s_{\text{след}}, a_1), Q(s_{\text{след}}, a_2) \ldots Q(s_{\text{след}}, a_N)], \tag{11.3}$$

где $s_{\text{след}}$ — состояние, в которое мы попадем, если выберем в состоянии s_i действие a. Приведенное уравнение, известное под названием *уравнения Беллмана*[1], описывает

[1] Сформулировано американским прикладным математиком Ричардом Э. Беллманом (1920–1984). См. его книгу Dynamic Programming, Princeton University Press, 1957. (*Беллман Р. Э.* Динамическое программирование. — М.: Изд-во иностр. литературы, 1960.)

на абстрактном языке то, как мы получили числа 6 и –0,6 для действий a_1 и a_2 в предыдущем простом примере. Попросту говоря, данное уравнение гласит:

- Q-значение выполнения действия a в состоянии s_i равно сумме двух составляющих:
  1) немедленного вознаграждения от выполнения действия a и...
  2) наилучшего (в смысле оптимального выбора действия в следующем состоянии) возможного Q-значения этого следующего состояния, умноженного на коэффициент дисконтирования.

Q-обучение возможно именно благодаря уравнению Беллмана, так что важно хорошо в нем разобраться. Ваш внутренний программист наверняка заметил его рекурсивную природу: все Q-значения в правой половине уравнения можно разложить далее на основе самого уравнения. Приведенный на рис. 11.10 пример завершился на втором шаге этой рекурсии, но в настоящих задачах MDP число шагов и состояний обычно намного больше, и графы состояний — действий — переходов могут даже содержать циклы. Но истинная красота и мощь уравнения Беллмана — в том, что с его помощью можно превратить задачу Q-обучения в задачу обучения с учителем даже для очень больших пространств состояний. В следующем разделе мы расскажем, почему это так.

11.3.3. Глубокая Q-сеть

Вручную описывать функцию $Q(s, a)$ непросто, так что лучше воспользоваться для нее глубокой нейронной сетью (упомянутой ранее в этом разделе DQN) и подобрать ее параметры путем обучения. Эта DQN получает на входе тензор, отражающий полное состояние среды, то есть положение на доске змейки, наблюдаемое агентом. Как видно из рис. 11.11, форма этого тензора — `[9, 9, 2]` (не считая измерения батчей). Первые два измерения соответствуют высоте и ширине игровой доски. Таким образом, этот тензор можно считать представлением всех клеток доски в виде битовой карты. Последнее измерение (2) представляет собой два канала, соответствующих змейке и фруктам соответственно. В частности, змейка кодируется в первом канале, причем ее голова обозначается 2, а тело — 1. Фрукт кодируется во втором канале, с помощью значения 1. В обоих каналах пустые клетки обозначаются 0. Учтите, что эти значения и количество каналов взяты более или менее «с потолка». Можно использовать и другие значения (например, 100 для головы змейки и 50 для ее тела, а можно и разбить голову и тело змейки на два канала), главное, сохранить три отдельных типа сущностей (голову змейки, тело змейки и фрукт).

Учтите также, что память в этом тензорном представлении состояния игры расходуется менее экономно, чем в описанном в предыдущем разделе JSON-представлении, состоящем из полей `s` и `f`, поскольку в него всегда включаются все клетки доски, вне зависимости от длины змейки. Мы будем использовать это неэкономичное представление лишь при обновлении весовых коэффициентов DQN с помощью обратного распространения ошибки. Кроме того, в каждый конкретный момент времени подобным образом представляется лишь небольшое число

(`batchSize`) состояний игры благодаря парадигме обучения по батчам, которую мы скоро обсудим.

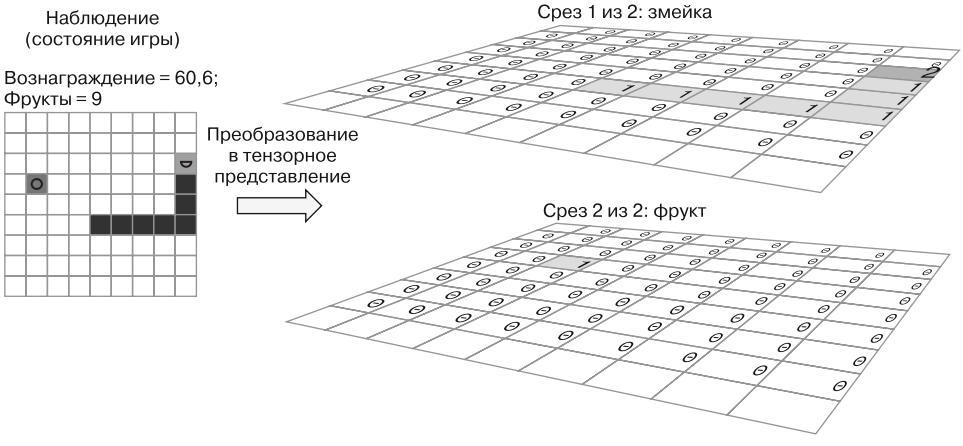


Рис. 11.11. Представление состояния доски игры «Змейка» в виде трехмерного тензора формы [9, 9, 2]

Код преобразования экономичного представления состояния доски в тензоры, подобные представленным на рис. 11.11, можно найти в функции `getStateTensor()` из файла `snake-dqn/snake_game.js`. Эта функция активно используется во время обучения DQN, но мы не станем приводить здесь подробности ее внутреннего устройства, поскольку она всего лишь механически присваивает значения элементам тензорного буфера в соответствии с расположением змейки и фруктов.

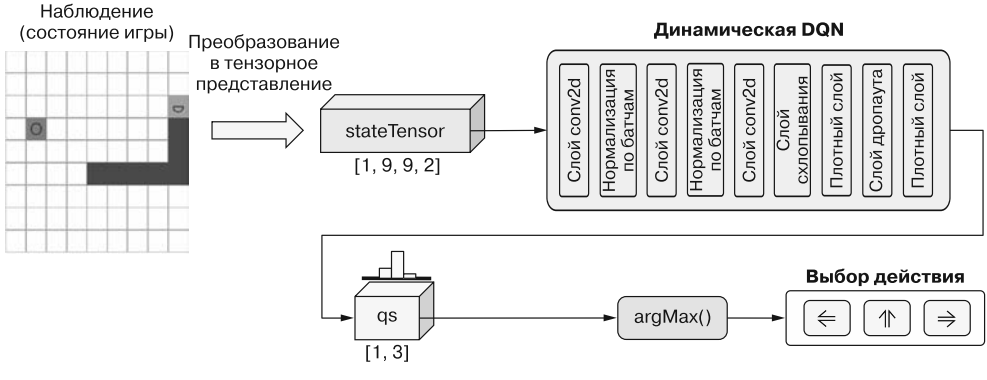


Рис. 11.12. Схематическая иллюстрация DQN, используемой для аппроксимации функции $Q(s, a)$ для задачи змейки

Наверное, вы обратили внимание, что входной формат `[height, width, channel]` идеально подходит для обработки сверточными сетями. Архитектура DQN представляет собой хорошо уже нам знакомую архитектуру сверточной сети. Код описания

топологии DQN приведен в листинге 11.5 (он представляет собой фрагмент из файла `snake-dqn/dqn.js`, из которого удалена часть кода обработки ошибок ради большей ясности). Как демонстрируют этот код и схема на рис. 11.12, наша сеть состоит из набора слоев conv2d, за которыми следует MLP. Для повышения возможностей обобщения DQN добавлены дополнительные слои, включая слой нормализации по батчам и слой дропаута. Форма выходного сигнала DQN: `[3]` (не считая измерения батчей). Три элемента выходного сигнала представляют собой предсказанные Q-значения соответствующих действий (поворот влево, движение прямо и поворот вправо). Таким образом, наша модель $Q(s, a)$ представляет собой нейронную сеть с состоянием среды в качестве входного сигнала и Q-значениями всех возможных действий для этого состояния в качестве выходного сигнала.

Листинг 11.5. Создание DQN для задачи змейки

```
export function createDeepQNetwork(h, w, numActions) {
  const model = tf.sequential();
  model.add(tf.layers.conv2d({
    filters: 128,
    kernelSize: 3,
    strides: 1,
    activation: 'relu',
    inputShape: [h, w, 2]
  }));
  model.add(tf.layers.batchNormalization());
  model.add(tf.layers.conv2d({
    filters: 256,
    kernelSize: 3,
    strides: 1,
    activation: 'relu'
  }));
  model.add(tf.layers.batchNormalization());
  model.add(tf.layers.conv2d({
    filters: 256,
    kernelSize: 3,
    strides: 1,
    activation: 'relu'
  }));
  model.add(tf.layers.flatten());
  model.add(tf.layers.dense({units: 100, activation: 'relu'}));
  model.add(tf.layers.dropout({rate: 0.25}));
  model.add(tf.layers.dense({units: numActions}));
  return model;
}
```

У DQN — типичная для сверточной сети архитектура, которая начинается с нескольких слоев conv2d

Форма входного сигнала соответствует тензорному представлению наблюдений агента, как показано на рис. 11.11

Для борьбы с переобучением и улучшения обобщаемости добавляем слой нормализации по батчам

Входящий в состав DQN многослойный перцептрон начинается со слоя схлопывания

Как и слой нормализации по батчам, слой дропаута служит для борьбы с переобучением

Давайте на минуту отвлечемся и задумаемся: почему вообще имеет смысл использовать в этой задаче в качестве функции $Q(s, a)$ нейронную сеть. Пространство состояний игры «Змейка» — дискретное, в отличие от непрерывного пространства состояний задачи балансировки шеста на тележке, выражаемого четырьмя числами с плавающей точкой. Таким образом, функцию $Q(s, a)$ можно, в принципе, реализовать в виде поисковой таблицы, в которой каждому возможному сочетанию позиции на доске и действия ставится в соответствие значение функции Q. Так по-

чему же мы предпочли DQN подобной поисковой таблице? Дело в том, что даже при относительно небольшом размере доски (9×9) число возможных позиций на ней слишком велико[1], что приводит к двум основным проблемам подхода с поисковой таблицей. Во-первых, такая громадная поисковая таблица не поместится в RAM системы. Во-вторых, даже если нам удастся создать систему с достаточным объемом памяти, обход всех состояний агентом во время обучения с подкреплением займет непозволительно много времени. Благодаря небольшому размеру (около 1 миллиона параметров) DQN решает первую (нехватка памяти) проблему. А благодаря способностям нейронных сетей к обобщению — вторую (время, необходимое на обход состояний). Как мы видели в предыдущих главах, нейронная сеть не обязательно должна видеть все возможные входные сигналы; она постепенно обучается интерполировать обучающие примеры данных посредством обобщения. Следовательно, при использовании DQN мы убиваем двух зайцев одним выстрелом.

11.3.4. Обучение глубокой Q-сети

Теперь у нас есть DQN для оценки Q-значений трех возможных действий на каждом шаге игры «Змейка». Для получения максимально возможного совокупного вознаграждения нам достаточно запустить DQN на основе наблюдений на каждом шаге и выбрать действие с максимальным Q-значением. Все готово? Нет, поскольку DQN пока что не обучена! Без должного обучения DQN содержит лишь заданные случайным образом начальные значения весовых коэффициентов и будет предлагать действия не лучше взятых «с потолка». Таким образом, мы свели задачу RL змейки к вопросу обучения DQN, который мы в этом разделе и обсудим. Процесс этот не совсем прост. Но не волнуйтесь: мы приведем множество схем, сопровождаемых фрагментами кода, для пошаговой иллюстрации алгоритма обучения.

---

[1] Приблизительный подсчет показывает, что количество возможных позиций на доске — как минимум порядка 10^{15}, даже если ограничить длину змейки 20 клетками. Например, рассмотрим змейку длиной 20. Во-первых, существует $9 \times 9 = 81$ возможных местоположений для головы змейки. А далее — четыре варианта расположения первого сегмента ее тела, три варианта расположения второго сегмента и т. д. Конечно, при некоторых вариантах размещения тела змейки вариантов будет меньше трех, но на порядок величин это особо не повлияет. Таким образом, можно приблизительно оценить число возможных размещений тела змейки длиной 20 как $81 \times 4 \times 3^{18} \approx 10^{12}$. А если учесть, что каждому размещению тела змейки соответствует 61 возможное местоположение фрукта, оценка количества возможных совместных размещений змейки и фрукта вырастает до 10^{14}. Аналогично можно оценить число вариантов и для меньшей длины тела змейки, от 2 до 19. Суммируя все оценки для длин от 2 до 20, получаем величину порядка 10^{15}. Количество пикселов в компьютерных играх, таких как Atari 2600, намного превышает число клеток на нашей доске для змейки, а потому еще меньше подходит для использования поисковой таблицы. Это одна из причин, почему для решения связанных с подобными компьютерными играми задач с помощью RL подходят DQN, как было показано в переломной статье 2015 года, написанной специалистами из DeepMind Владимиром Мнихом и др.

Основная идея обучения глубокой Q-сети

Мы будем обучать нашу DQN, заставляя ее соответствовать уравнению Беллмана. В случае удачи DQN будет отражать как немедленные вознаграждения, так и оптимальные дисконтированные будущие вознаграждения.

Как это реализовать? Нам понадобится множество примеров пар «входной — выходной сигнал», где роль входного сигнала играет состояние и фактически выполненное действие, а роль выходного сигнала — «правильное» (целевое) значение Q. Для вычисления примеров входных сигналов необходимо текущее состояние s_i и предпринимаемое на этом состоянии действие, a_j, которые оба доступны в истории игры. Для вычисления целевого значения Q необходима информация о немедленном вознаграждении r_i и следующем состоянии s_{i+1}, также доступная в истории игры. Можно использовать для вычисления целевого Q-значения r_i и s_{i+1} путем применения уравнения Беллмана, подробности чего мы вскоре опишем. Далее можно вычислить разницу предсказанного DQN Q-значения и целевого Q-значения из уравнения Беллмана и использовать это значение в качестве функции потерь. Мы будем минимизировать потери (в смысле метода наименьших квадратов) с помощью обычного обратного распространения ошибки и градиентного спуска. Внутренние механизмы, благодаря которым это возможно и работает эффективно, довольно сложны, но основная идея проста. Нам требуется оценка значения функции Q для принятия хороших решений. Мы знаем, что наша оценка Q должна соответствовать вознаграждениям среды и уравнению Беллмана, так что воспользуемся градиентным спуском. Все просто!

Воспроизводимая память: скользящий набор данных для обучения DQN

Наша DQN представляет собой привычную нам сверточную сеть, реализованную в виде экземпляра `tf.LayersModel` в TensorFlow.js. Первое, что приходит на ум в смысле того, как ее обучить: вызвать ее метод `fit()` или метод `fitDataset()`. Однако тут использовать этот обычный подход мы не можем, поскольку у нас нет маркированного набора данных с наблюдаемыми состояниями и соответствующими Q-значениями. Пока DQN не обучена, никак нельзя узнать Q-значения. Если бы у нас был метод, возвращающий истинные Q-значения, можно было бы воспользоваться им в марковском процессе принятия решений и все. Таким образом, если ограничиться традиционным подходом обучения с учителем, мы столкнемся с проблемой курицы/яйца: без обученной DQN нельзя оценить Q-значения; без хорошей оценки Q-значений нельзя обучить DQN. Алгоритм RL, с которым мы собираемся вас сейчас познакомить, как раз и поможет нам решить эту проблему.

Если конкретнее, при нашем методе агент будет играть в игру случайным образом (по крайней мере сначала) и запоминать происходящее на каждом ходе игры. Реализовать игру случайным образом легко с помощью генератора случайных чисел. А запоминание — с помощью структуры данных под названием *«воспроизводимая память»* (replay memory). На рис. 11.13 показано, как она работает. На каждом шаге игры в ней сохраняется пять элементов.

1. s_i — наблюдение текущего состояния на шаге i (позиция на доске).
2. a_i — выполняемое на текущем шаге действие (выбираемое либо DQN, как показано на рис. 11.12, или путем случайной выборки).
3. r_i — получаемое на этом шаге немедленное вознаграждение.
4. d_i — булев флаг, указывающий, завершается ли игра сразу после текущего шага. Из него видно, что воспроизводимая память служит не только для отдельного эпизода игры, а объединяет результаты множества игровых эпизодов. По завершении предыдущей игры алгоритм обучения просто начинает новую и продолжает добавлять новые записи в воспроизводимую память.
5. s_{i+1} — наблюдение со следующего шага, если d_i равен `false` (если d_i равен `true`, в качестве «заполнителя» сохраняется пустое значение).

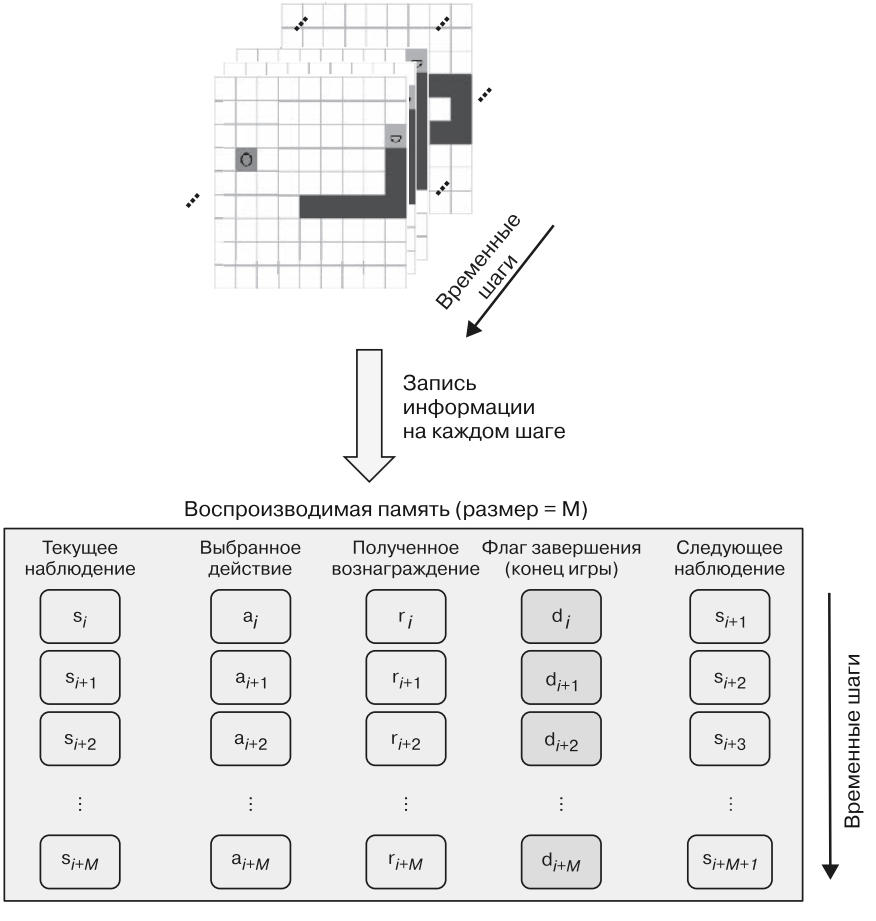


Рис. 11.13. Использование воспроизводимой памяти во время обучения DQN. На каждом шаге в конец воспроизводимой памяти добавляется пять элементов данных, выбираемых во время обучения DQN

Эти элементы данных служат входными для обучения DQN на основе обратного распространения ошибки. Воспроизводимую память можно считать набором данных для обучения DQN. Однако она отличается от наборов данных в обучении с учителем тем, что обновляется по ходу обучения. Длина воспроизводимой памяти — фиксированная, M (по умолчанию в нашем примере кода M = 10 000). При вставке в ее конец записи $(s_i, a_i, r_i, d_i, s_{i+1})$ после следующего хода игры из ее начала удаляется старая запись, в результате чего сохраняется фиксированная длина воспроизводимой памяти. Таким образом, воспроизводимая память гарантированно содержит информацию о произошедшем на последних M шагах обучения, помимо того, что помогает решить проблемы нехватки памяти. Полезно всегда обучать DQN на последних записях игры. Почему? А потому, что, когда DQN уже немного обучилась и начала «понимать, что к чему» в игре, обучать ее на старых записях игры, например, с начала обучения нежелательно, ведь они могут включать «наивные» ходы, более не уместные или не благоприятствующие дальнейшему обучению сети.

Реализующий воспроизводимую память код очень прост, его можно найти в файле `snake-dqn/replay_memory.js`. Мы не станем описывать его подробно, за исключением двух общедоступных методов, `append()` и `sample()`.

- С помощью метода `append()` можно вставить новую запись в конец воспроизводимой памяти.
- Вызов `sample(batchSize)` выбирает случайным образом `batchSize` записей из воспроизводимой памяти. Выборка этих записей производится совершенно равномерно и включает в общем случае записи из нескольких различных эпизодов. Мы будем использовать метод `sample()` для извлечения обучающих батчей во время вычисления функции потерь и последующего обратного распространения ошибки, как мы вскоре увидим.

Эпсилон-жадный алгоритм: компромисс между исследованием и использованием

Агенту, пробующему различные случайные ходы, рано или поздно повезет, и он наткнется на удачные (съест фрукт-другой в игре «Змейка»), что может принести пользу в качестве начального толчка в процессе обучения. На самом деле этот способ — единственный, ведь агенту неизвестны правила игры. Но если агент и дальше будет вести себя случайным образом, то долго такой процесс обучения не продлится, ведь выбираемые случайно варианты приводят к игровой гибели, а также потому, что достичь определенных продвинутых состояний можно только посредством целой череды удачных ходов.

Это само воплощение дилеммы «исследовать или использовать» применительно к игре «Змейка». Мы уже сталкивались с этой дилеммой в примере с тележкой и шестом, где она решалась с помощью метода градиентного спуска по стратегиям, благодаря постепенному росту детерминизма полиномиальной выборки в ходе обучения. В игре «Змейка» такой возможности у нас нет, поскольку действия выбираются на основе не `tf.multinomial()`, а исходя из максимального Q-значения для всех возможных действий. Поэтому мы решим эту проблему путем параметризации степени случайности процесса выбора действий и постепенного уменьшения этого

параметра случайности. Если конкретно, мы будем использовать так называемую *эпсилон-жадную стратегию* (epsilon-greedy policy), которую можно выразить в псевдокоде следующим образом:

```
x = Выборка случайного числа равномерно из интервала от 0 до 1
if x < epsilon:
  Выбрать действие случайным образом
else:
  qValues = DQN.предсказать(наблюдение)
  Выбрать действие, соответствующее максимальному элементу qValues
```

Эта логика применяется на каждом шаге обучения. Чем больше (ближе к 1) значение эпсилон, тем вероятнее, что действие будет выбрано случайным образом. И напротив, чем значение эпсилон меньше (ближе к 0), тем выше вероятность того, что действие будет выбрано на основе предсказанных DQN Q-значений. Случайный выбор действий можно считать исследованием среды, в то время как выбор действий с целью максимизации Q-значения называют *жадным*. Отсюда и возникло название *эпсилон-жадная*.

Как показано в листинге 11.6, настоящий код TensorFlow.js, реализующий эпсилон-жадный алгоритм в примере snake-dqn, очень напоминает предыдущий псевдокод. Данный код представляет собой фрагмент из файла `snake-dqn/agent.js`.

Листинг 11.6. Фрагмент код примера snake-dqn, реализующий эпсилон-жадный алгоритм

```
let action;
const state = this.game.getState();
if (Math.random() < this.epsilon) {
  action = getRandomAction();
} else {
  tf.tidy(() => {
    const stateTensor =
        getStateTensor(state,
                       this.game.height,
                       this.game.width);
    action = ALL_ACTIONS[
        this.onlineNetwork.predict(
            stateTensor).argMax(-1).dataSync()[0]];
  });
}
```

Исследование: выбор действий случайным образом

Состояние игры в виде тензора

Жадная стратегия: получаем от DQN предсказанные Q-значения и находим индекс действия, соответствующего максимальному Q-значению

Эпсилон-жадная стратегия обеспечивает компромисс между необходимостью исследования вначале и необходимостью стабильного поведения впоследствии, для чего постепенно уменьшает значение эпсилон с относительно большого до близкого (но не равного) нулю. В нашем примере snake-dqn эпсилон линейно уменьшается с 0,5 до 0,01 за первые 1×105 шагов обучения. Учтите, что мы не уменьшаем эпсилон до самого нуля, поскольку даже на продвинутых этапах обучения агента нужна некоторая доля исследования для поиска новых интересных ходов. В задачах RL, основанных на эпсилон-жадной стратегии, начальное и конечное значения эпсилон играют роль настраиваемых гиперпараметров, как и скорость уменьшения его значения.

После закладки фундамента нашего алгоритма Q-обучения в виде эпсилон-жадной стратегии, перейдем к подробностям обучения DQN.

Извлечение предсказанных Q-значений

Хотя мы пытаемся решить задачу RL с помощью нового подхода, но все равно хотели бы свести наш алгоритм к обучению с учителем, чтобы воспользоваться привычным методом обратного распространения ошибки для обновления весовых коэффициентов DQN. Для подобной постановки задачи нужны три вещи.

- Предсказанные Q-значения.
- «Истинные» Q-значения. Обратите внимание, что слово «истинные» взято в кавычки, поскольку на самом деле не существует способа получить эталонные Q-значения, так что эти значения — просто наилучшие из доступных нам на конкретном этапе алгоритма обучения оценок $Q(s, a)$. Поэтому мы будем называть их вместо этого целевыми Q-значениями.
- Функция потерь, возвращающая по предсказанному и целевому Q-значениям число, количественно выражающее расхождение между ними.

В этом подразделе мы обсудим извлечение предсказанных Q-значений из воспроизводимой памяти. В последующих двух подразделах мы поговорим, соответственно, о том, как получить целевые Q-значения и функцию потерь. А когда у нас будут все три эти составляющие, задача RL змейки превратится, по существу, в простую задачу обратного распространения ошибки.

На рис. 11.14 показано, как предсказанные Q-значения извлекаются из воспроизводимой памяти на одном из шагов обучения DQN. Эту схему следует изучать совместно с реализующим ее кодом в листинге 11.7, чтобы проще было в них разобраться.

В частности, мы выбираем случайным образом `batchSize` (`N = 128` по умолчанию) записей из воспроизводимой памяти. Как уже описывалось ранее, каждая запись состоит из пяти элементов. Для получения предсказанных Q-значений нам нужны только первые два. Эти первые элементы, состоящие из N наблюдений состояния, преобразуются вместе в тензор. Динамический DQN обрабатывает этот тензор батча наблюдений и возвращает предсказанные Q-значения (`qs` на схеме и в коде). Однако `qs` включает Q-значения не только фактически выбранных действий, но и невыбранных. В нашем обучении мы хотим игнорировать Q-значения для невыбранных действий, поскольку не существует способа узнать их целевые Q-значения. Именно здесь нам пригодится второй элемент воспроизводимой памяти.

Этот второй элемент содержит фактически выбранные действия в тензорном представлении (`actionTensor` на схеме и в коде). Далее `actionTensor` используется для выбора нужных нам элементов `qs`. Этот шаг, показанный на рисунке в прямоугольнике «Выбор фактических действий», реализуется с помощью трех функций TensorFlow.js: `tf.oneHot()`, `mul()` и `sum()` (см. последнюю строку листинга 11.17). Его нельзя свести к простым срезам тензора, поскольку различные действия могут выбираться на различных ходах игры. Код в листинге 11.7 представляет собой фрагмент метода `SnakeGameAgent.trainOnReplayBatch()` из файла `snake-dqn/agent.js`, в котором опущено несколько мелких нюансов для простоты.

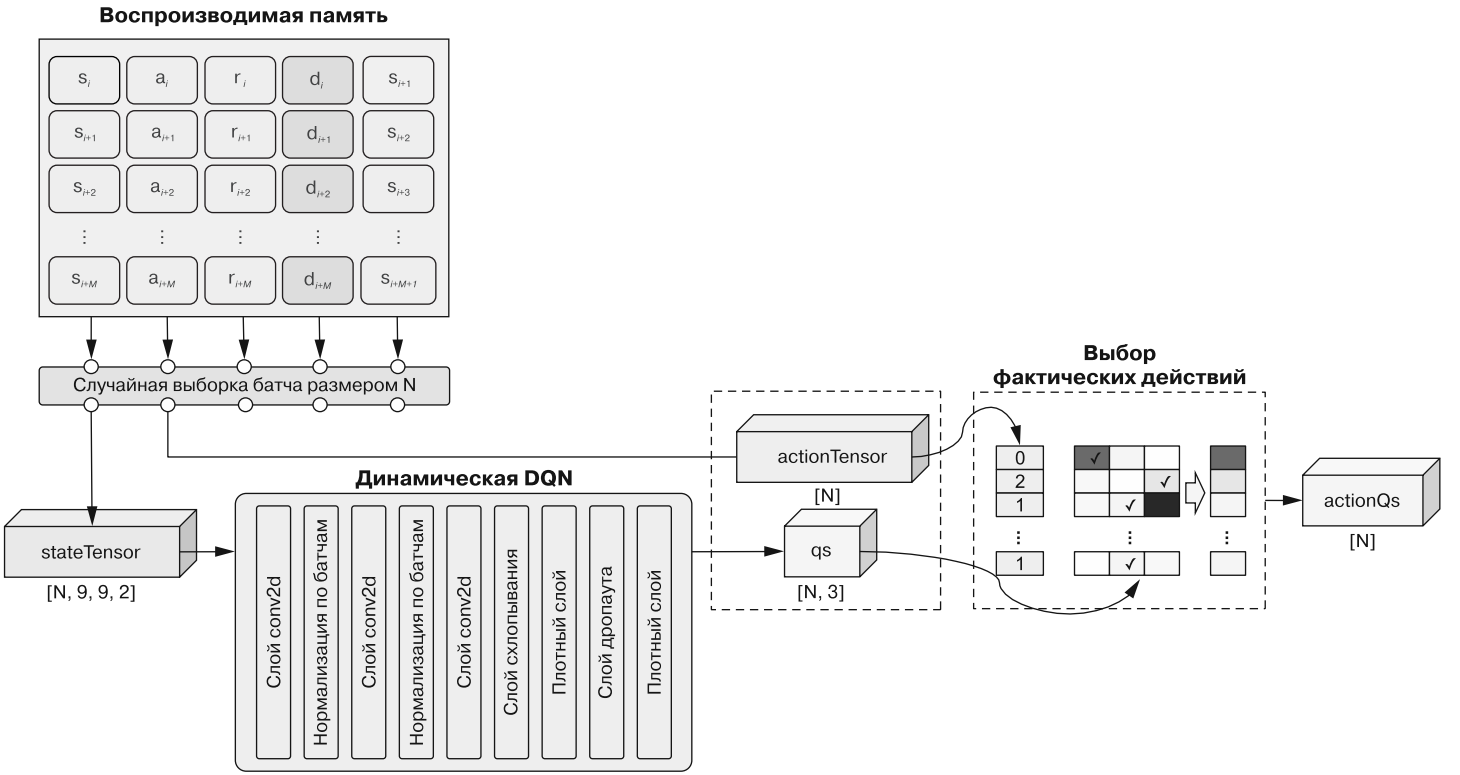
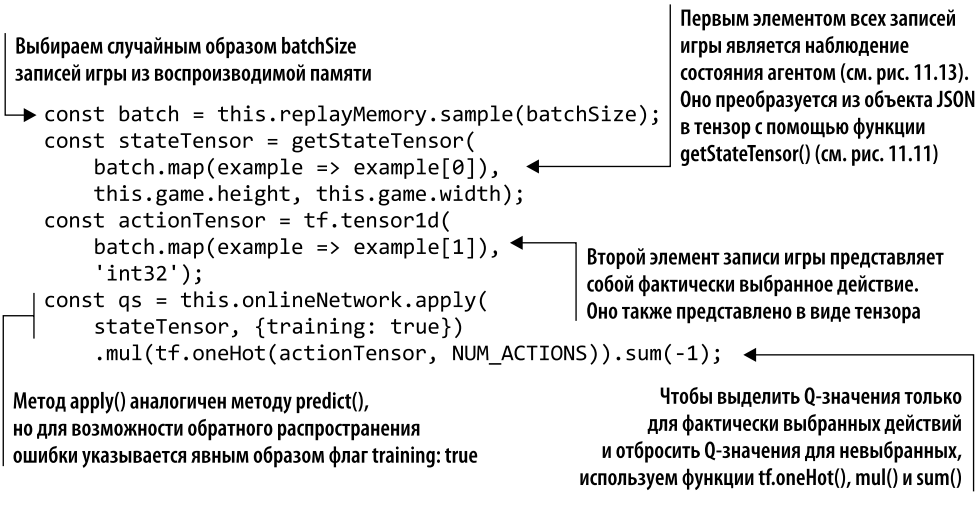


Рис. 11.14. Получение предсказанных Q-значений из воспроизводимой памяти и динамической DQN. Это первый из двух компонентов обучения с учителем в алгоритме обучения DQN. Результат этого технологического процесса, actionQs, — предсказанные DQN Q-значения — представляет собой один из двух аргументов вместе с targetQs, участвующих в вычислении потерь MSE. См. технологический процесс вычисления targetQs на рис. 11.15

Листинг 11.7. Извлечение батча предсказанных Q-значений из воспроизводимой памяти



В результате этих операций мы получаем тензор `actionQs` формы `[N]`, где `N` — размер батча. Он и представляет собой искомое Q-значение, то есть предсказанное значение $Q(s, a)$ для состояния s, в котором мы находились, и фактически выполненного действия a. Далее мы поговорим о получении целевых Q-значений.

Извлечение целевых Q-значений: применяем уравнение Беллмана

Получить целевые Q-значения несколько сложнее, чем предсказанные. Здесь мы воспользуемся на практике теоретическим уравнением Беллмана. Напомним, что уравнение Беллмана описывает Q-значение пары «состояние — действие» на основе двух элементов: 1) немедленного вознаграждения и 2) максимального Q-значения, которого можно добиться из состояния следующего хода (дисконтированное в соответствии с коэффициентом). Проще получить первый из них, поскольку он доступен в виде третьего элемента воспроизводимой памяти, что показано на рис. 11.15 в виде `rewardTensor`.

Для вычисления второго (максимального Q-значения для следующего хода) нам понадобится наблюдение состояния со следующего хода. К счастью, наблюдение состояния со следующего хода хранится в пятом элементе воспроизводимой памяти. Мы берем наблюдение состояния следующего хода нашего выбранного случайным образом батча, преобразуем его в тензор, пропускаем через копию DQN — так называемую *целевую DQN* (рис. 11.15) — и получаем оценки Q-значений для состояний следующего хода. После этого вызываем функцию `max()` по последнему измерению (измерению батчей) и получаем максимальные Q-значения, достижимые из состояния следующего хода (тензор `nextMaxQTensor` в листинге 11.8). Согласно уравнению Беллмана умножаем это максимальное значение на коэффициент дисконтирования (γ на рис. 11.15 и `gamma` в листинге 11.8) и в сочетании с немедленным вознаграждением получаем целевые Q-значения (`targetQs` на схеме и в коде).

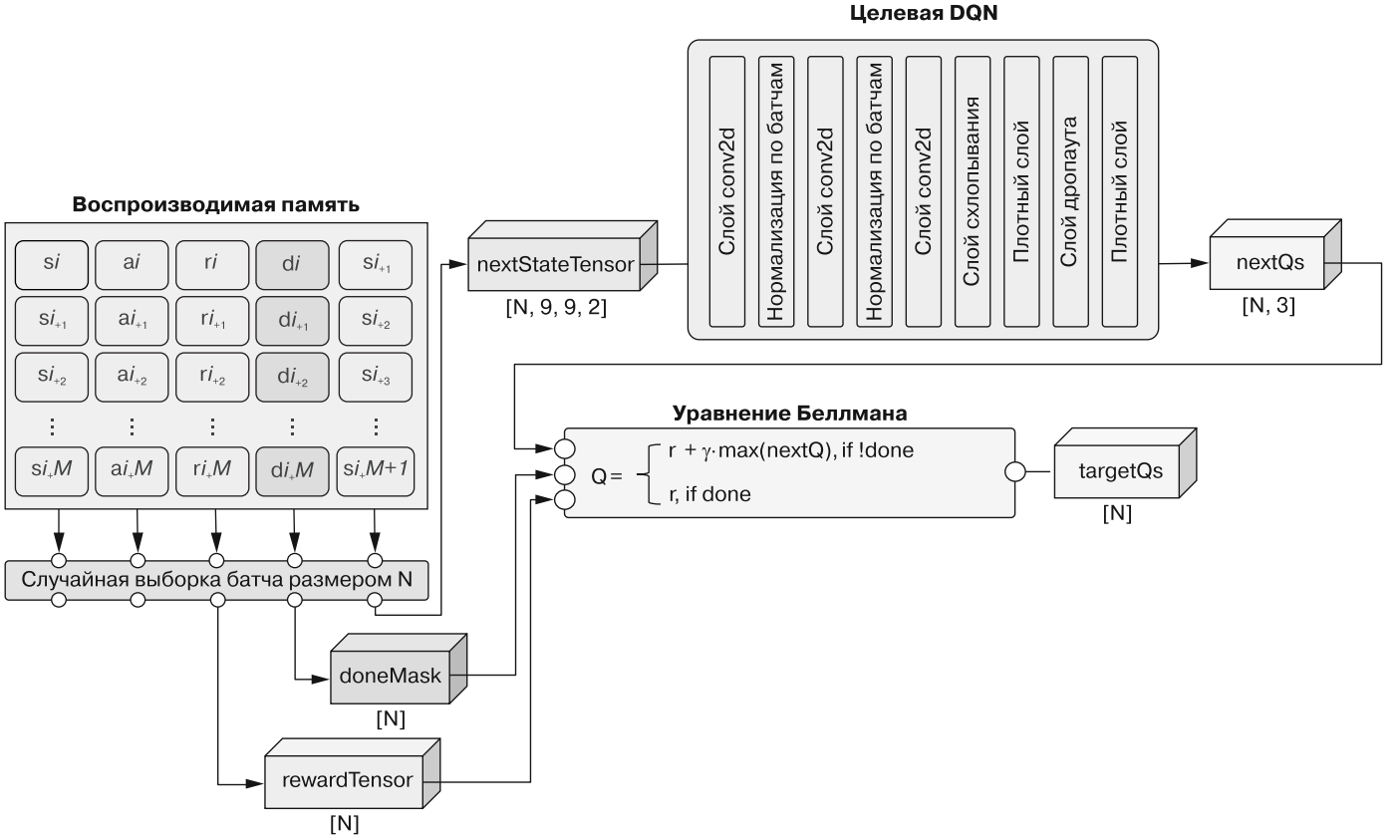


Рис. 11.15. Получение целевых Q-значений (targetQs) из воспроизводимой памяти и целевой DQN. Относящиеся к воспроизводимой памяти и выборке батча части на этом рисунке совпадают с рис. 11.14. Его следует изучать вместе с кодом в листинге 11.8. Это второй компонент обучения с учителем в алгоритме обучения DQN. targetQs играют роль, аналогичную истинным меткам в задачах обучения с учителем в предыдущих главах (например, известные истинные метки в примерах MNIST или известные истинные значения температуры в примере Jena-weather). Ключевую роль в вычислении targetQs играет уравнение Беллмана. Вместе с целевым DQN это уравнение связывает Q-значения текущего хода и Q-значения следующего хода, благодаря чему дает возможность вычислить значения targetQs

Учтите, что Q-значения следующего шага существуют только в том случае, если текущий ход не является последним ходом эпизода игры (то есть змейка на нем не погибает). В противном случае правая сторона уравнения Беллмана включает только компонент для немедленного вознаграждения, как показано на рис. 11.15, что соответствует тензору `doneMask` в листинге 11.8. Код в листинге 11.8 представляет собой фрагмент метода `SnakeGameAgent.trainOnReplayBatch()` из файла `snake-dqn/agent.js`, в котором опущено несколько мелких нюансов для простоты.

Листинг 11.8. Извлечение батча целевых («истинных») Q-значений из воспроизводимой памяти

```
const rewardTensor = tf.tensor1d(
    batch.map(example => example[2]));
const nextStateTensor = getStateTensor(
    batch.map(example => example[4]),
    this.game.height, this.game.width);
const nextMaxQTensor =
    this.targetNetwork.predict(nextStateTensor)
    .max(-1);
const doneMask = tf.scalar(1).sub(
    tf.tensor1d(batch.map(example => example[3]))
        .asType('float32'));
const targetQs =
    rewardTensor.add(nextMaxQTensor.mul(
        doneMask).mul(gamma));
```

Третий элемент записи воспроизводимой памяти содержит значение немедленного вознаграждения

Четвертый элемент записи содержит наблюдение следующего состояния, преобразуемое в тензорное представление

Целевая DQN применяется к тензору следующего состояния, в результате чего получаются Q-значения для всех действий на следующем ходе

doneMask равно 0 для завершающих игру ходов и 1 для прочих ходов

Для вычисления целевых Q-значений используется уравнение Беллмана

Для выяснения максимально возможного вознаграждения на следующем ходе используется функция max(). Это правая сторона уравнения Беллмана

Как вы могли заметить, хитрость этого алгоритма глубокого Q-обучения заключается в использовании двух экземпляров DQN — *динамической* DQN и *целевой* DQN соответственно. Динамическая DQN отвечает за вычисление предсказанных Q-значений (см. рис. 11.14 в предыдущем подразделе). Также именно с ее помощью выбирается действие змейки, когда эпсилон-жадный алгоритм выбирает жадный (никакого исследования) подход. Именно отсюда и название «динамическая». Напротив, целевая DQN используется только для вычисления целевых Q-значений, как мы только что видели, потому и называется целевой. Вы спросите, зачем две DQN вместо одной? Чтобы разорвать нежелательные петли обратной связи, могущие привести к неустойчивости процесса обучения.

Создает динамическую DQN и целевую DQN одна и та же функция — `createDeepQNetwork()` (см. листинг 11.5). Они представляют собой две глубокие сверточные сети с одинаковой топологией. Следовательно, их наборы слоев и весовых коэффициентов также совпадают. Периодически (каждые 1000 ходов при настройках по умолчанию snake-dqn) значения весов копируются из динамической DQN в целевую. Благодаря этому целевая DQN соответствует текущему состоянию динамической DQN. Без такой синхронизации целевая DQN быстро устареет по сравнению с динамической и будет мешать процессу обучения, выдавая плохие оценки наилучших Q-значений для следующего хода в уравнении Беллмана.

Функция потерь для предсказания Q-значений и обратного распространения ошибки

Воспользуемся уже знакомой вам функцией потерь `meanSquaredError` для вычисления расхождения полученных предсказанных и целевых Q-значений (рис. 11.16). Таким образом, нам удалось свести процесс обучения DQN к задаче регрессии, схожей с предыдущими примерами, например Boston-housing и Jena-weather. Сигнал рассогласования функции потерь `meanSquaredError` служит движущей силой обратного распространения ошибки, а обновление динамической DQN производится на основе полученных в результате обновлений весовых коэффициентов.

Схема на рис. 11.16 включает части, уже показанные на рис. 11.14 и 11.15. На ней не только эти части собраны воедино, но и добавлены новые прямоугольники и стрелки для функции потерь `meanSquaredError` и основанного на ней обратного распространения ошибки (см. правую нижнюю часть схемы). Таким образом, на ней приведена полная картина алгоритма глубокого Q-обучения нашего агента для игры «Змейка».

Код в листинге 11.9, содержащий внутренности метода `trainOnReplayBatch()` класса `SnakeGameAgent` из файла `snake-dqn/agent.js`, играющего ключевую роль в нашем алгоритме RL, тесно связан со схемой на рис. 11.16. В этом методе описана функция потерь для вычисления `meanSquaredError` между предсказанными и целевыми Q-значениями. Далее он вычисляет градиенты `meanSquaredError` относительно весовых коэффициентов динамической DQN с помощью функции `tf.variableGrads()` (в разделе Б.4 вы найдете подробный обзор функций вычисления градиентов TensorFlow.js, в частности, и `tf.variableGrads()`). Далее, с помощью оптимизатора вычисленные градиенты применяются для обновления весовых коэффициентов DQN, подталкивая тем самым динамическую DQN в сторону более точных оценок Q-значений. После повторения этого миллионы раз получается DQN, обеспечивающий неплохие игровые результаты в игре «Змейка». Отвечающая за вычисление целевых Q-значений (`targetQs`) часть кода из следующего листинга уже приводилась в листинге 11.8.

Вот и все, что мы хотели рассказать о внутреннем устройстве алгоритма глубокого Q-обучения. Запустить обучение на основе этого алгоритма в среде Node.js можно с помощью следующей команды:

```
yarn train --logDir /tmp/snake_logs
```

При наличии настроенного должным образом GPU с поддержкой CUDA можете добавить в эту команду флаг `--gpu` для ускорения обучения. В результате указания флага `--logDir` данная команда во время обучения заносит в журнал в каталоге журналов TensorFlow.js следующие метрики: 1) скользящее среднее совокупных вознаграждений за 100 последних эпизодов игры (`cumulativeReward100`); 2) скользящее среднее количества «съеденных» за 100 последних эпизодов игры фруктов (`eaten100`); 3) значение параметра исследования (`epsilon`) и 4) скорость обучения, выражаемую в ходах в секунду (`framesPerSecond`). Просмотреть эти журналы можно, запустив TensorFlow.js с помощью следующих команд и перейдя в браузере по HTTP URL клиентской части TensorFlow.js (по умолчанию http://localhost:6006):

```
pip install tensorboard tensorboard --logdir /tmp/snake_logs
```

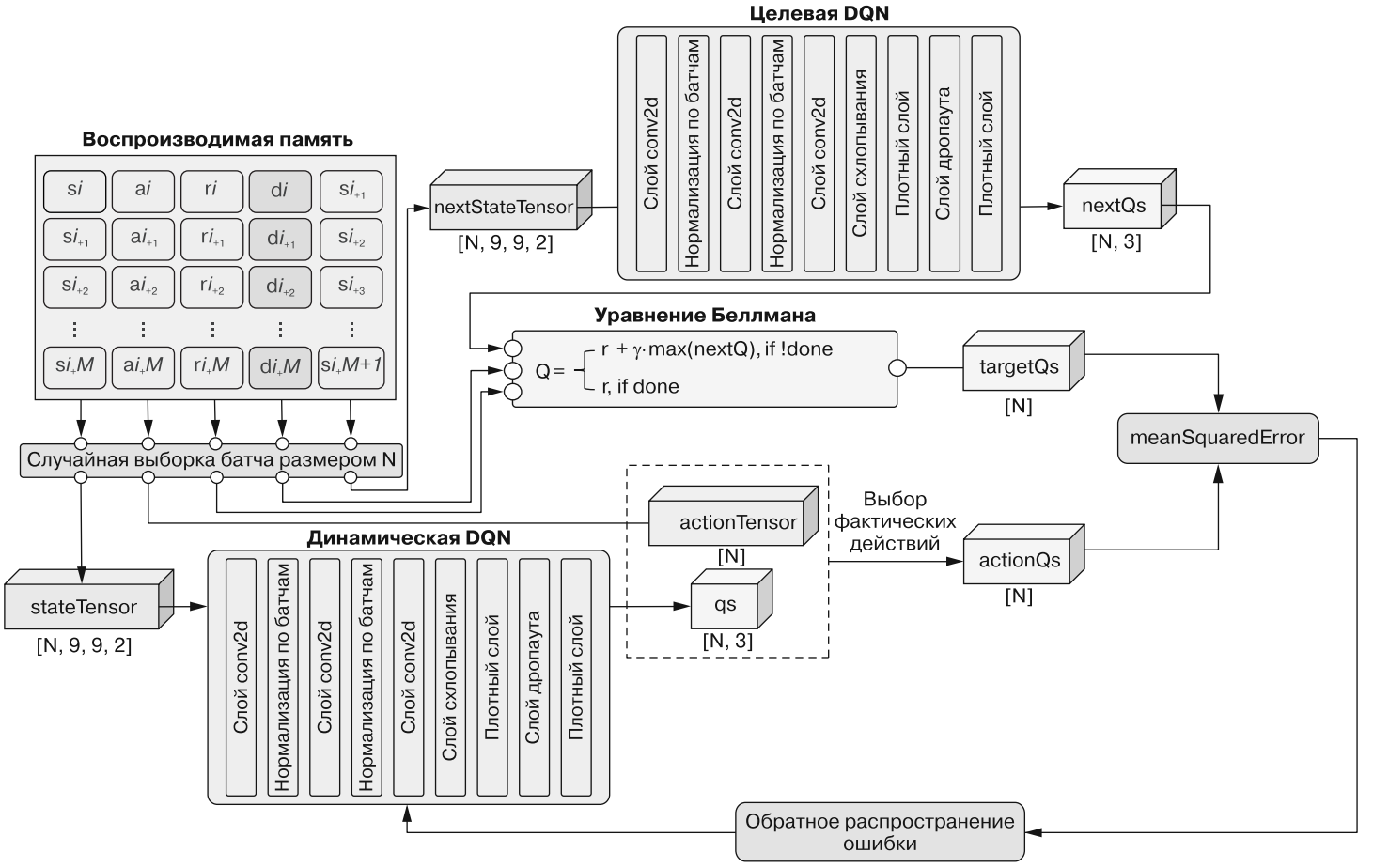


Рис. 11.16. Объединяем actionQs и targetQs для вычисления погрешности предсказания meanSquaredError динамической DQN и дальнейшего обновления весовых коэффициентов путем обратного распространения ошибки. Большую часть этой схемы вы уже видели на рис. 11.14 и 11.15. Были добавлены только функция потерь meanSquaredError и шаг обратного распространения ошибки на ее основе в нижней правой части схемы

Листинг 11.9. Основная функция, отвечающая за обучение DQN

```
trainOnReplayBatch(batchSize, gamma, optimizer) {
  const batch =this.replayMemory.sample(batchSize);
  const lossFunction = () => tf.tidy(() => {
    const stateTensor = getStateTensor(
        batch.map(example => example[0]),
                  this.game.height,
                  this.game.width);
    const actionTensor = tf.tensor1d(
        batch.map(example => example[1]), 'int32');
    const qs = this.onlineNetwork
        .apply(stateTensor, {training: true})
        .mul(tf.oneHot(actionTensor, NUM_ACTIONS)).sum(-1);

    const rewardTensor = tf.tensor1d(batch.map(example => example[2]));
    const nextStateTensor = getStateTensor(
        batch.map(example => example[4]),
                  this.game.height, this.game.width);
    const nextMaxQTensor =
        this.targetNetwork.predict(nextStateTensor).max(-1);
    const doneMask = tf.scalar(1).sub(
        tf.tensor1d(batch.map(example => example[3])).asType('float32'));
    const targetQs =
        rewardTensor.add(nextMaxQTensor.mul(doneMask).mul(gamma));
    return tf.losses.meanSquaredError(targetQs, qs);
  });

  const grads = tf.variableGrads(
      lossFunction, this.onlineNetwork.getWeights());
  optimizer.applyGradients(grads.grads);
  tf.dispose(grads);
}
```

Получаем случайный батч примеров данных из буфера воспроизводимой памяти

lossFunction возвращает скалярное значение, используемое для обратного распространения ошибки

Предсказанные Q-значения

Целевые Q-значения, вычисленные на основе уравнения Беллмана

Мерой расхождения предсказанных и целевых Q-значений служит MSE

Вычисляет градиент lossFunction относительно весовых коэффициентов динамической DQN

С помощью оптимизатора обновляет весовые коэффициенты на основе вычисленных градиентов

На рис. 11.17 приведен набор типичных кривых процесса обучения из журналов. Как это часто встречается в обучении с подкреплением, кривые `cumulativeReward100` и `eaten100` демонстрируют колебания. После нескольких часов обучения модель достигнет наилучшего значения `cumulativeReward100` 70–80 и наилучшего значения `eaten100` примерно 12.

Обучающий сценарий также сохраняет модель по относительному пути `./models/dqn` при всяком достижении нового наилучшего значения `cumulativeReward100`. При выполнении команды `yarn watch` сохраненная модель выдается в веб-версии клиентской части. Клиентская часть отображает Q-значения, предсказываемые DQN на каждом ходе игры (см. рис. 11.18). Применяемую во время обучения эпсилон-жадную стратегию при игре после обучения заменяет всегда жадная стратегия. В качестве действия змейки всегда выбирается действие, соответствующее максимальному Q-значению (например, 33,9 — двигаться прямо — на рис. 11.18). Из таких графиков понятно на интуитивном уровне, как DQN играет в игру.

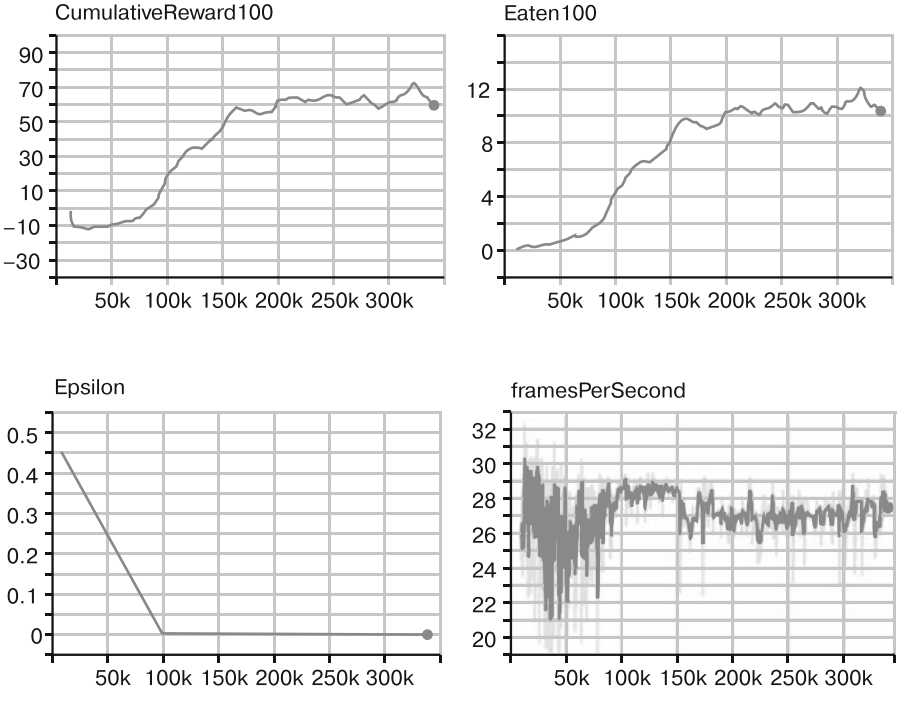


Рис. 11.17. Примеры журналов с процесса обучения snake-dqn в tfjs-node. На рисунках показаны: 1) cumulativeReward100 — скользящее среднее совокупного вознаграждения за последние 100 игр; 2) eaten100 — скользящее среднее количества «съеденных» за последние 100 игр фруктов; 3) epsilon — значение эпсилон, по которому можно видеть динамику эпсилон-жадной стратегии и 4) framesPerSecond — мера скорости обучения

Из поведения змейки можно сделать несколько интересных наблюдений. Во-первых, количество фактически съеденных змейкой фруктов на демонстрации в клиентской части (~18) в среднем превышает значения кривой `eaten100` из журналов обучения (~12) из-за отказа от эпсилон-жадной стратегии, а значит, и исключения случайных действий во время игры. Напомним, что на поздних этапах обучения DQN значение эпсилон остается небольшим, но все же не равным нулю (см. третий блок на рис. 11.17). Вызываемые этим значением эпсилон случайные действия порой приводят к преждевременной гибели змейки — такова цена исследовательского поведения. Во-вторых, змейка выработала интересную стратегию перемещения к боковым границам и углам доски перед приближением к фрукту, даже когда фрукт располагается возле центра доски. Такая стратегия помогает змей-

Рис. 11.18. Q-значения, предсказываемые обученной DQN, отображаются в числовом виде и отрисовываются различными оттенками зеленого в веб-версии клиентской части игры

ке при относительно большой ее длине (скажем, в районе 10–18) снизить вероятность натолкнуться на свое тело. Это не так уж плохо, но и не так уж хорошо тоже, поскольку существуют более грамотные стратегии, выработать которые змейке не удалось. Например, змейка часто замыкается в круг, когда ее длина превышает 20. Но на большее алгоритм из snake-dqn не способен. Для дальнейшего усовершенствования агента змейки необходимо видоизменить эпсилон-жадный алгоритм так, чтобы подстегнуть змейку при большой длине искать лучшие ходы[1]. В текущем алгоритме исследовательское начало оказывается слишком слабо к моменту, когда змейка вырастает до размеров, требующих умелого лавирования вокруг ее собственного тела.

На этом завершается наш обзор методики DQN для обучения с подкреплением. В основе нашего алгоритма лежит статья «Управление на уровне человека с помощью обучения с подкреплением»[2], в которой исследователи из DeepMind впервые показали, как сочетание возможностей глубоких нейронных сетей и RL позволяет машинам успешно играть во многие компьютерные игры наподобие Atari 2600. Приведенное здесь решение snake-dqn представляет собой упрощенную версию алгоритма DeepMind. Например, наш DQN анализирует наблюдение только с текущего шага, в то время как алгоритм DeepMind объединяет во входном сигнале DQN текущее наблюдение с наблюдениями из нескольких прошлых ходов. Но наш пример захватывает саму суть этой революционной методики — а именно, использование глубокой сверточной сети в качестве мощного средства аппроксимации функций для оценки зависящих от состояния ценностей действий и обучения ее с помощью MDP и уравнения Беллмана. Последующие достижения исследователей RL, например покорение таких игр, как го и шахматы, основаны на аналогичном союзе глубоких нейронных сетей и традиционных, не связанных с глубоким обучением методов обучения с подкреплением.

Материалы для дальнейшего изучения

- *Sutton R. S., Barto A. G.* Reinforcement Learning: An Introduction, A Bradford Book, 2018. (*Саттон Р. С., Барто Э. Дж.* Обучение с подкреплением. — М.: ДМК-Пресс, 2020.)
- Конспект лекций Дэвида Сильвера по обучению с подкреплением, прочитанных в Университетском колледже Лондона: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.
- *Zai A., Brown B.* Deep Reinforcement Learning in Action, Manning Publications: www.manning.com/books/deep-reinforcement-learning-in-action.
- *Lapan M.* Deep Reinforcement Learning Hands-On: Apply Modern RL Methods, with Deep Q-networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More. — Packt Publishing, 2018. (*Лапань М.* Глубокое обучение с подкреплением. AlphaGo и другие технологии. — СПб.: Питер, 2020.)

---

1 Например, см. https://github.com/carsonprindle/OpenAIExam2018.

2 *Mnih V. et al.* Human-Level Control through Deep Reinforcement Learning // Nature, 2015. Vol. 518. Pp. 529–533. www.nature.com/articles/nature14236/.

Упражнения

1. В примере тележки с шестом использовалась сеть стратегий, состоящая из скрытого слоя с 128 нейронами, в соответствии с настройками по умолчанию. Как влияет этот гиперпараметр на обучение, в основе которого лежит градиентный спуск по стратегиям? Попробуйте поменять его на небольшое значение, например 4 или 8, и сравните полученную кривую обучения (кривую зависимости среднего количества ходов на игру от итерации) с аналогичной кривой при размере скрытого слоя по умолчанию. Какой вывод можно сделать из этого о связи между разрешающими возможностями модели и эффективностью оценки ею наилучшего действия?
2. Мы уже упоминали, что одно из преимуществ решения задач наподобие тележки с шестом с помощью машинного обучения — экономия затрат человеческого труда. А именно, при неожиданном изменении среды не нужно выяснять, *как* именно она изменилась, и перерабатывать физические уравнения. Вместо этого достаточно повторного обучения агента. Убедитесь, что это так, по следующему алгоритму. Во-первых, запустите пример cart-pole из исходного кода, а не с веб-страницы на сервере. Обучите соответствующую сеть стратегий с помощью обычного подхода. Во-вторых, поменяйте значение `this.gravity` в файле `cart-pole/cart_pole.js` на новое (скажем, 12, чтобы притвориться, что мы перенесли нашу тележку с шестом на другую планету, с большей силой тяжести, чем на Земле!). Запустите страницу снова, загрузите обученную на первом шаге сеть стратегий и попробуйте ее в работе. Можно ли сказать, что она работает намного хуже, чем раньше, просто из-за повышенной силы тяжести? Наконец, обучите сеть стратегий на протяжении еще нескольких итераций. Начинает ли она снова играть лучше (приспосабливается к новой среде)?
3. (Упражнение, посвященное марковским процессам принятия решений и уравнению Беллмана.) Представленный в подразделе 11.3.2 и на рис. 11.10 пример MDP был прост в смысле полного его детерминизма, поскольку в переходах между состояниями и соответствующих вознаграждениях никакой стохастичности не было. Но многие реальные задачи лучше описываются с помощью стохастических (случайных) MDP. В стохастическом марковском процессе состояние, в котором оказывается агент, и получаемое им вследствие действия вознаграждение описывается вероятностным распределением. Например, как демонстрирует рис. 11.19, если агент выполняет действие a_1 в состоянии s_1, то с вероят-

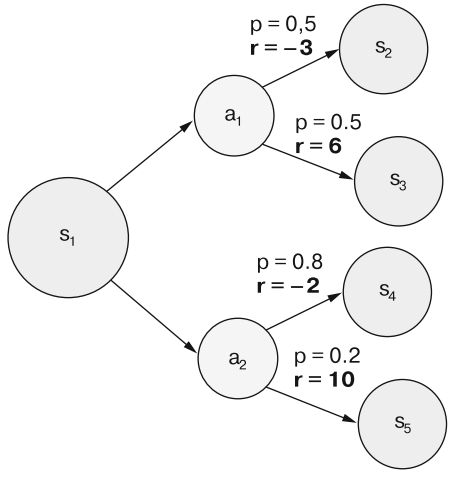


Рис. 11.19. Схема марковского процесса принятия решений в первой части упражнения 3

ностью 0,5 оказывается в состоянии s_2, и с вероятностью 0,5 же — в состоянии s_3. Соответствующие этим двум переходам вознаграждения различаются. В подобных стохастических MDP агенту необходимо учитывать случайность посредством вычисления *ожидаемого* будущего вознаграждения. Ожидаемое будущее вознаграждение представляет собой взвешенное среднее значение всех возможных вознаграждений, в котором веса выражают вероятности. Попробуйте применить этот вероятностный подход и оценить Q-значения для a_1 и a_2 в состоянии s_1 на рисунке. Согласно полученному вами ответу какое действие лучше выполнить в состоянии s_1: a_1 или a_2?

Теперь рассмотрим несколько более сложный стохастический марковский процесс принятия решений, состоящий из нескольких шагов (рис. 11.20). В этом чуть более сложном случае необходимо использовать рекурсивное уравнение Беллмана для учета наилучших возможных будущих вознаграждений после первого действия, которые сами являются стохастическими. Учтите, что иногда эпизод завершается после первого шага, а иногда — после второго. Можно ли определить, какое действие лучше будет выполнить в состоянии s_1? Для этой задачи используйте коэффициент дисконтирования 0,9.

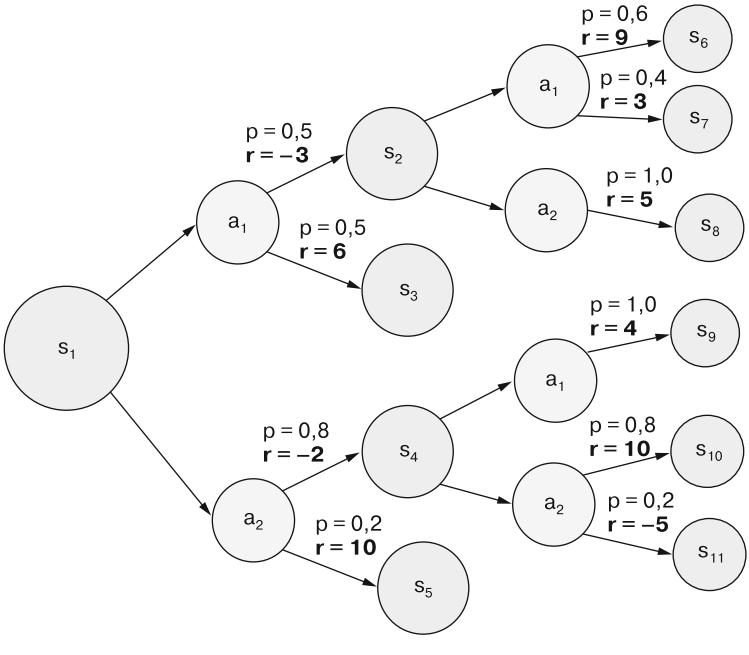


Рис. 11.20. Схема марковского процесса принятия решений во второй части упражнения 3

4. В примере snake-dqn мы применяли эпсилон-жадную стратегию для получения оптимального соотношения исследования/использования. При настройках по умолчанию эпсилон уменьшается от начального значения 0,5 до конечного

значения 0,01 и остается на этом уровне. Попробуйте поменять конечное значение эпсилон на большее (например, 0,1) или меньшее (например, 0) и посмотрите, насколько хорошо при этом обучается агент змейки. Можете пояснить полученную разницу с точки зрения роли, которую играет эпсилон?

Резюме

- Как тип машинного обучения, обучение с подкреплением связано с принятием оптимальных решений. В задачах RL агент обучается так выбирать действия в среде, чтобы максимизировать метрику *совокупного вознаграждения*.
- В отличие от обучения с учителем в RL нет маркированных обучающих наборов данных. Вместо этого агенту приходится усваивать, какие действия хороши при различных обстоятельствах, пробуя их случайным образом.
- Мы изучили два часто используемых типа алгоритмов обучения с подкреплением: методы на основе стратегий (на примере удержания в равновесии шеста в тележке) и методы на основе Q-значений (на примере игры «Змейка»).
- Стратегия — это алгоритм, с помощью которого агент выбирает действие на основе наблюдений текущего состояния. Стратегия может быть заключена в нейронной сети, принимающей в качестве входного сигнала наблюдение состояния и генерирующей в виде выходного сигнала выбор действия. Подобные нейронные сети называются *сетями стратегий*. В задаче тележки с шестом мы применяли градиентный спуск по стратегиям и метод REINFORCE для обновления и обучения сети стратегий.
- В отличие от методов на основе стратегий при Q-обучении для оценки ценностей действий при заданном наблюдаемом состоянии применяется модель вида *Q-сеть*. В примере snake-dqn мы показали, как в этом качестве может выступать глубокая сверточная сеть и как обучить ее с помощью MDP, уравнения Беллмана и *воспроизводимой памяти*.

Часть IV

Резюме и заключительное слово

Заключительная часть книги состоит из двух глав. Глава 12 охватывает проблемы, встречающиеся пользователям TensorFlow.js при развертывании моделей в среде промышленной эксплуатации. В ней мы обсудим рекомендуемые практики, благодаря которым разработчик может быть уверен в безошибочности модели, методики для уменьшения размера моделей и повышения эффективности их работы, а также спектр поддерживаемых TensorFlow.js сред развертывания. Глава 13 подытоживает всю книгу, резюмируя ключевые понятия, технологические процессы и методики.

12 Тестирование, оптимизация и развертывание моделей

При участии Яника Ассогбы, Пинь Ю и Ника Кригера

В этой главе

- Важность тестирования и мониторинга кода машинного обучения и практические рекомендации.
- Оптимизация обученных в TensorFlow.js или преобразованных в TensorFlow.js моделей для ускорения загрузки и выполнения вывода.
- Развертывание моделей TensorFlow.js на различных платформах и в разных средах от браузерных расширений до мобильных приложений и от приложений для настольных компьютеров до одноплатных компьютеров.

Как мы упоминали в главе 1, машинное обучение отличается от традиционной инженерии разработки ПО автоматизацией обнаружения правил и эвристических алгоритмов. Из предыдущих глав эта уникальная особенность машинного обучения уже должна быть вам ясна. Впрочем, модели машинного обучения и окружающий их код остаются кодом, функционирующим как часть общего комплекса программных средств. Чтобы гарантировать надежную и эффективную работу моделей машинного обучения, необходимы те же меры предосторожности, что и для обычного, не связанного с машинным обучением кода.

Эта глава посвящена практической стороне применения TensorFlow.js для машинного обучения как части общего стека ПО. В разделе 12.1 мы обсудим жизненно важный, хотя часто игнорируемый вопрос тестирования и мониторинга кода и моделей машинного обучения. В разделе 12.2 приведены утилиты и уловки для снижения размера обученных моделей и объема потребляемых ими ресурсов, ускорения их скачивания и выполнения, критически важных для развертывания моделей на стороне как клиента, так и сервера. В завершающем разделе мы пройдемся по различным средам, в которых можно развернуть созданные с помощью TensorFlow.js модели, и обсудим в ходе этого характерные преимущества, ограничения и стратегии каждого из вариантов развертывания.

К концу данной главы вы познакомитесь с рекомендуемыми практиками тестирования, оптимизации и развертывания моделей глубокого обучения в TensorFlow.js.

12.1. Тестирование моделей TensorFlow.js

До сих пор мы говорили о проектировании, создании и обучении моделей МО. Теперь же мы собираемся заняться некоторыми вопросами, возникающими при развертывании уже обученных моделей, начиная с тестирования — как кода машинного обучения, так и сопутствующего ему кода, не относящегося к машинному обучению. Основные проблемы, с которыми вы столкнетесь, если захотите охватить модель и процесс ее обучения тестами, — размер модели, время обучения и недетерминированное поведение во время обучения (например, инициализация весовых коэффициентов случайными значениями и некоторые операции нейронных сетей наподобие дропаута). По мере перехода от отдельной модели к полноценному приложению вы также столкнетесь с разнообразными асимметриями и расхождениями ветвей кода для обучения и выполнения вывода, проблемами контроля версий моделей и расхождениями в данных для разных аудиторий пользователей. Вам предстоит увидеть, что для общей надежности системы машинного обучения и уверенности в ней тестирование должно сопровождаться продуманным мониторингом.

Один из ключевых вопросов: «Как происходит управление версиями вашей модели?» В большинстве случаев модель настраивается и обучается до тех пор, пока не будет достигнута удовлетворительная степень безошибочности на проверочном наборе данных, после чего никакой дальнейшей подстройки не требует. В ходе обычного процесса создания модели она не пересоздается и не обучается заново. Вместо этого можно внести топологию и усвоенные весовые коэффициенты модели в систему контроля версий в виде скорее большого двоичного объекта (BLOB), а не артефакта текста/кода. Изменение окружающего модель кода не должно приводить к изменению номера версии модели. Аналогично обучение модели заново и внесение ее в репозиторий не должно требовать изменения исходного кода, не относящегося к модели.

Какие аспекты системы машинного обучения необходимо охватить тестами? По нашему мнению, ответ: все. Рисунок 12.1 поясняет почему. Типовая система, простирающаяся от входных данных до готовой к развертыванию обученной модели,

состоит из нескольких основных компонентов. Некоторые из них, напоминающие обычный, не относящийся к машинному обучению код, подходят для обычного модульного тестирования, в то время как другие демонстрируют характерные для машинного обучения особенности, а потому требуют специального тестирования или мониторинга. Но главный вывод: нельзя игнорировать или недооценивать важность тестирования лишь потому, что речь идет о системе машинного обучения. Скорее, наоборот, модульное тестирование еще важнее для кода машинного обучения, возможно, даже больше, чем для разработки обычного программного обеспечения, поскольку алгоритмы машинного обучения нередко труднее для понимания, чем обычные, не связанные с машинным обучением алгоритмы. Они часто терпят неудачу из-за плохих входных данных, из-за чего возникают малозаметные и трудные для отладки проблемы, единственной защитой от которых является тестирование и мониторинг. В следующих подразделах мы подробнее обсудим различные фрагменты рис. 12.1.

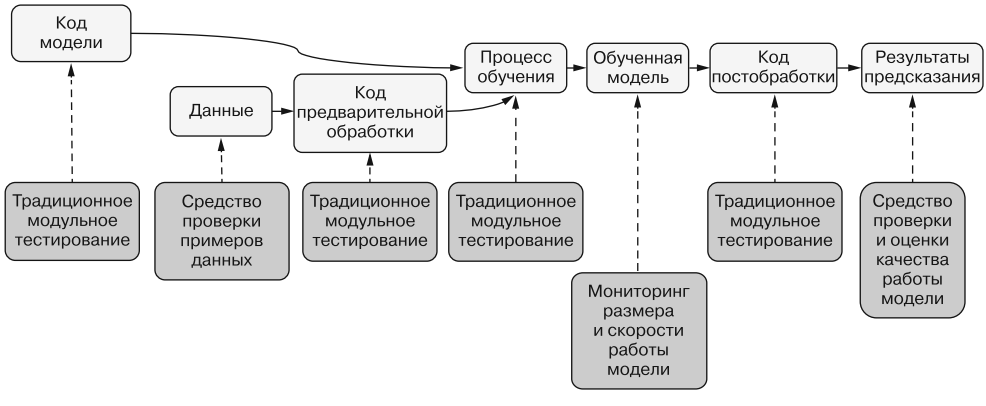


Рис. 12.1. Охват готовой к промышленной эксплуатации системы машинного обучения тестами и мониторингом. Верхняя половина схемы включает основные компоненты типового конвейера создания и обучения модели МО. Нижняя половина демонстрирует применимые к каждому из компонентов меры традиционного модульного тестирования. Некоторые из компонентов подходят для традиционных методик тестирования: код создания и обучения модели, а также код предварительной обработки входных данных модели и постобработки ее выходных результатов. Для прочих компонентов необходимы более специализированные (применительно к машинному обучению) меры тестирования и мониторинга, включающие проверку качества данных в примерах, мониторинг размеров и скорости вывода обученной модели, а также тонкую проверку и оценку выполненных ею предсказаний

12.1.1. Традиционное модульное тестирование

Как и в случае обычных, не МО проектов, в основе вашего набора тестов должны лежать надежные простые модульные тесты. Однако для создания модульных тестов моделей машинного обучения необходимо учесть некоторую специфику. Как вы видели в предыдущих главах, для количественного выражения итогового качества

работы модели после успешного обучения и настройки гиперпараметров часто используются такие метрики, как степень безошибочности на проверочном наборе данных. Подобные метрики оценки играют важную роль в мониторинге модели инженерами-людьми, но не подходят для автоматического тестирования. Бывает заманчиво добавить тест для контроля того, что определенная метрика превышает определенное пороговое значение (например, AUC для задачи бинарной классификации превышает 0,95 или MSE для задачи регрессии — меньше 0,2). Однако подобные операторы контроля на основе пороговых значений следует использовать очень осторожно, а то и вообще избегать их, поскольку они очень ненадежны. Процесс обучения модели включает несколько источников случайности, в том числе начальные значения весовых коэффициентов и перетасовку обучающих примеров, вследствие чего результаты обучения модели различаются от запуска к запуску. Еще одним источником неустойчивости может быть изменение набора данных (например, из-за регулярного добавления новых данных). Поэтому выбрать подходящее пороговое значение весьма непросто. При слишком мягком пороговом значении вы рискуете пропустить реальные проблемы. При слишком жестком — рискуете столкнуться с большим числом ложных срабатываний.

Отключить случайную составляющую программ TensorFlow.js обычно можно с помощью вызова функции `Math.seedrandom()` перед созданием и выполнением модели. Например, следующая строка кода задает конкретное начальное значение для случайной инициализации весовых коэффициентов, перетасовки данных и слоев дропаута, так что последующее обучение модели становится детерминированным:

```
Math.seedrandom(42);
```

42 — просто выбранное произвольным образом фиксированное начальное значение для генератора случайных чисел

Этот прием очень удобен при написании тестов, включающих операторы контроля для значений потерь или метрик.

Однако даже при детерминированном задании начального значения генератора случайных чисел тестирования одного только `model.fit()` и аналогичных вызовов недостаточно для хорошего покрытия тестами кода машинного обучения. Как и в случае других плохо поддающихся тестированию частей кода, следует стараться полностью охватить модульными тестами окружающий модель код, который можно легко протестировать с их помощью, а для самой модели искать альтернативные решения. Весь код загрузки данных, предварительной обработки, постобработки выходных сигналов модели и прочие вспомогательные методы обычно легко поддаются обычному тестированию. Кроме того, небольшого количества нестрогих тестов самой модели — формы входных и выходных сигналов, например, в совокупности со стилем тестирования «проверить, что модель не генерирует исключение после первого же шага обучения» — вполне достаточно в качестве минимальной тестовой обвязки модели, чтобы чувствовать себя в безопасности при рефакторинге. (Как вы, наверное, заметили при экспериментах с примерами кода из предыдущих глав, для тестирования в tfjs-examples мы использовали фреймворк тестирования Jasmine, но вы можете использовать любой фреймворк модульного тестирования, какой только нравится вам и вашей команде).

В качестве иллюстрации применения этого на практике рассмотрим тесты для примера анализа тональностей из главы 9. Просматривая код этого примера, вы могли заметить файлы `data_test.js`, `embedding_test.js`, `sequence_utils_test.js` и `train_test.js`. Первые три из этих файлов охватывают тестами не относящийся к модели код и выглядят в точности как обычные модульные тесты. Их наличие повышает нашу уверенность в правильности формата поступающих на вход модели во время обучения и выполнения вывода данных, а также допустимости наших операций с ними.

Последний из файлов в этом списке касается самой модели машинного обучения, а потому заслуживает немного больше нашего внимания. Фрагмент из него приведен в листинге 12.1.

Листинг 12.1. Модульные тесты API модели — форм ее входных и выходных сигналов, а также ее обучаемости

```
describe('buildModel', () => {
 it('flatten training and inference', async () => {
    const maxLen = 5;
    const vocabSize = 3;
    const embeddingSize = 8;
    const model = buildModel('flatten', maxLen, vocabSize, embeddingSize);
    expect(model.inputs.length).toEqual(1);
    expect(model.inputs[0].shape).toEqual([null, maxLen]);
    expect(model.outputs.length).toEqual(1);
    expect(model.outputs[0].shape).toEqual([null, 1]);
    model.compile({
      loss: 'binaryCrossentropy',
      optimizer: 'rmsprop',
      metrics: ['acc']
    });
    const xs = tf.ones([2, maxLen])
    const ys = tf.ones([2, 1]);
    const history = await model.fit(xs, ys, {
      epochs: 2,
      batchSize: 2
    });
    expect(history.history.loss.length).toEqual(2);
    expect(history.history.acc.length).toEqual(2);

    const predictOuts = model.predict(xs);
    expect(predictOuts.shape).toEqual([2, 1]);
    const values = predictOuts.arraySync();
    expect(values[0][0]).toBeGreaterThanOrEqual(0);
    expect(values[0][0]).toBeLessThanOrEqual(1);
    expect(values[1][0]).toBeGreaterThanOrEqual(0);
    expect(values[1][0]).toBeLessThanOrEqual(1);
  });
});
```

Убеждаемся, что формы входного и выходного сигналов модели — такие, как нужно

Очень краткое обучение модели — выполняется быстро, но степень безошибочности будет очень низкой

Проверяет выдачу метрик для каждого шага обучения в качестве сигнала, подтверждающего факт обучения

Выполнение предсказания на основе модели с упором на проверку правильности API

Проверяем, входит ли предсказание в диапазон возможных ответов; сверять с фактическим значением смысла нет, ведь обучение было очень кратким, а потому сходимость не гарантируется

Этот тест охватывает очень много всего, так что разобьем его на части. Сначала мы создаем модель, используя вспомогательную функцию. В ходе этого теста нас не интересует структура модели, мы обращаемся с ней как с «черным ящиком». Контролируем формы входных и выходных сигналов:

```
expect(model.inputs.length).toEqual(1);
expect(model.inputs[0].shape).toEqual([null, maxLen]);
expect(model.outputs.length).toEqual(1);
expect(model.outputs[0].shape).toEqual([null, 1]);
```

Подобные тесты могут уловить проблемы неправильной идентификации измерения батчей — регрессия или классификация, форма выходного сигнала и т. д. Далее мы компилируем и обучаем модель в течение очень маленького числа шагов. Наша цель — просто убедиться в обучаемости модели, безошибочность, устойчивость и сходимость нас пока что не волнуют:

```
const history = await model.fit(xs, ys, {epochs: 2, batchSize: 2})
expect(history.history.loss.length).toEqual(2);
expect(history.history.acc.length).toEqual(2);
```

Этот фрагмент кода также проверяет выдачу при обучении требуемых метрик для анализа, ведь если бы мы обучали ее по-настоящему, то хотели бы следить за ходом обучения и степенью безошибочности полученной в его результате модели. Наконец, пробуем простой вывод:

```
const predictOuts = model.predict(xs);
expect(predictOuts.shape).toEqual([2, 1]);
const values = predictOuts.arraySync();
expect(values[0][0]).toBeGreaterThanOrEqual(0);
expect(values[0][0]).toBeLessThanOrEqual(1);
expect(values[1][0]).toBeGreaterThanOrEqual(0);
expect(values[1][0]).toBeLessThanOrEqual(1);
```

Мы не сверяемся с каким-то конкретным результатом предсказания, ведь они могут меняться в зависимости от случайных начальных значений весовых коэффициентов или возможных будущих модификаций архитектуры модели. Мы проверяем просто, что получили предсказание и его значение входит в ожидаемый диапазон, в данном случае от 0 до 1.

Главный урок, который можно из этого извлечь: модель всегда должна успешно проходить данный тест, вне зависимости от изменения внутренней архитектуры модели, главное, чтобы не менялся входной/выходной API. Если же тест не пройден, значит, в модели есть проблемы. Это легкие и быстрые тесты, обеспечивающие сильную уверенность в правильности API, подходящие для присоединения к любым распространенным точкам подключения тестов.

12.1.2. Тестирование с помощью «золотых значений»

В предыдущем разделе мы говорили о модульном тестировании, для которого не требуется контроля порогового значения метрики или устойчивости/сходимости обучения. Рассмотрим теперь виды тестирования полностью обученных моделей, начиная

с проверки предсказаний для конкретных точек данных. Возможно, у вас найдутся какие-нибудь очевидные примеры данных для тестирования. Например, в случае программы для обнаружения объектов входное изображение с красивым большим котом должно быть отмечено как таковое; в случае средства анализа тональностей фрагмент текста с явно отрицательным отзывом покупателя должен также быть маркирован соответствующим образом. Подобные «правильные ответы» на заданные входные сигналы модели мы будем называть *«золотыми значениями»* (golden values). Если слепо следовать парадигме традиционного модульного тестирования, легко попасть в ловушку тестирования обученных моделей машинного обучения на «золотых значениях». В конце концов, мы хотели бы, чтобы хорошо обученное средство обнаружения объектов всегда отмечало изображение кошки как изображение кошки, правда? Не совсем. Тестирование на основе «золотых значений» может привести к проблемам при машинном обучении, поскольку при этом мы посягаем на разбиение набора данных на обучающий, проверочный и контрольный.

При наличии представительной выборки для обучающего и контрольного наборов данных и подходящей целевой метрики (безошибочность, полнота и т. д.) почему один пример данных должен считаться правильнее другого? При обучении модели МО важна степень безошибочности на обучающем и контрольном наборах данных в целом. Предсказания для отдельных примеров данных могут варьироваться в зависимости от выбранных гиперпараметров и начальных значений весовых коэффициентов. Если у вас есть примеры данных, которые обязательно необходимо классифицировать правильно и которые легко выявить, то почему бы не найти их прежде, чем требовать от модели МО их классифицировать и не обработать их с помощью обычного кода, не использующего машинное обучение? Подобные примеры данных порой применяются в системах обработки естественного языка, где подмножество входных запросов (например, часто встречающиеся и легко выявляемые запросы) автоматически перенаправляются для обработки модулю, в котором не применяется машинное обучение, в то время как за остальные запросы отвечает модель машинного обучения. Помимо экономии вычислительных ресурсов, эту часть кода легко протестировать с помощью традиционного модульного тестирования. И хотя может показаться, что добавление слоя бизнес-логики до (или после) МО-предиктора — просто лишний труд, оно предоставляет точки доступа для принудительной коррекции предсказаний. Кроме того, именно там можно добавить мониторинг или журналирование, которые явно понадобятся, когда вашу утилиту начнут использовать более широко. После этого вступления по очереди рассмотрим три основные причины для использования «золотых значений».

Подобные тесты на основе «золотых значений» часто применяют для сквозного тестирования системы — для проверки того, что выдаст система, получив на входе необработанные входные данные. При этом система МО обучается, через код обычной последовательности операций конечного пользователя запрашивается предсказание, и ответ возвращается пользователю. Все аналогично модульному тестированию из листинга 12.1, но система машинного обучения работает в связке с остальным приложением. Можно написать аналогичный листингу 12.1 тест, для которого фактический результат предсказания не будет важен, и на самом деле такой

тест будет надежнее. Однако очень заманчиво и логично будет связать его с парой пример «данных/предсказание», чтобы разработчики, возвращаясь к этому тесту позднее, сразу все понимали.

Здесь и начинаются проблемы — нам нужен пример данных, предсказание для которого известно и должно быть правильно, иначе сквозной тест не будет пройден. Поэтому мы добавим менее масштабный тест для проверки этого предсказания на части охватываемого сквозным тестом конвейера. Теперь, если сквозной тест не пройден, а этот меньший тест пройден успешно, можно будет ограничить поле поиска ошибки взаимодействиями между основной моделью машинного обучения и остальными частями конвейера (например, кодом ввода/обработки данных или постобработки). Если же оба теста не пройдены, значит, нарушен инвариант «пример/предсказание». В этом случае ценность тестов скорее диагностическая, но имеет смысл при таком двойном непрохождении тестов выбрать новый пример данных для кодирования, а не обучать всю модель заново.

Еще один распространенный источник тестирования на основе «золотых значений» — различные бизнес-требования. Допустим, безошибочность для какого-либо легко определимого подмножества примеров данных должна быть выше, чем для остальных. В этом случае, как уже упоминалось ранее, имеет смысл добавить перед или после модели слой бизнес-логики для обработки подобных примеров. Впрочем, можете поэкспериментировать с *заданием весов для примеров данных* (example weighting), при котором некоторые примеры считаются важнее прочих при вычислении общих метрик качества работы модели. Что не гарантирует правильность работы модели в целом, но подталкивает модель в сторону более точных результатов для этих примеров. Если реализация такого слоя бизнес-логики доставляет трудности, поскольку не получается легко заранее определить, какие свойства входных данных приводят к этим особым случаям, может понадобиться воспользоваться второй моделью, которая нужна лишь для того, чтобы понять, требуется ли принудительная коррекция. В этом случае применяется ансамбль моделей, а бизнес-логика выбирает нужное действие на основе сочетания предсказаний от двух слоев.

Последний из случаев — получение от пользователя сообщения о программной ошибке с примером данных, на котором модель выдает неправильный результат. Если он неправилен, исходя из бизнес-требований, мы возвращаемся к предыдущему сценарию. Если же он просто попадает в процент ошибок кривой эффективности модели, мы мало что можем сделать. Все в пределах приемлемой эффективности обученного алгоритма; какое-то количество ошибок бывает у всех моделей. Можно разве что добавить пару «пример данных/правильное предсказание» в обучающий/проверочный/контрольный набор данных по ситуации в надежде сгенерировать в будущем лучшую модель, но использовать «золотые значения» для модульного тестирования не следует.

Единственное исключение: если модель неизменна — весовые коэффициенты модели и ее архитектура внесены в систему контроля версий и не обновляются в тестах. Тогда вполне уместно использовать «золотые значения» для тестирования выходных сигналов основанной на модели системы вывода, поскольку ни модель, ни примеры данных не подвергаются изменениям. Подобная система вывода включает не только модель, а и, например, части, отвечающие за предварительную обработку входных

данных перед вводом их в модель, а также преобразование выходных сигналов модели в форму, более подходящую для дальнейших систем. В этом случае модульные тесты помогают гарантировать корректность подобной логики предварительной и постобработки.

Еще один допустимый вид применения «золотых значений» выходит за рамки модульного тестирования: мониторинг качества работы модели (но не в качестве модульного тестирования) по мере ее видоизменения. Мы расскажем об этом подробнее, когда будем обсуждать средства проверки и оценки модели в следующем разделе.

12.1.3. Соображения по поводу непрерывного обучения

Во многих системах машинного обучения новые обучающие данные поступают довольно регулярно (каждый день или каждую неделю). Иногда можно использовать журналы за предыдущий день для генерации новых, более актуальных обучающих данных. В подобных системах модель необходимо часто обучать заново, на основе самых свежих доступных данных. Существует мнение, что в подобных случаях возможности устаревшей модели снижаются. С течением времени входные данные модели понемногу перестают соответствовать распределению, которому соответствовали во время ее обучения, и характеристики качества ее работы ухудшаются. В качестве примера представьте себе утилиту для рекомендации одежды, обученную зимой, а предсказания выполняющую летом.

Достаточно приступить к изучению систем непрерывного обучения, и вы обнаружите, что в конвейер входит большое число дополнительных компонентов. Всестороннее обсуждение их выходит за рамки данной книги, скажем только, что источником дополнительных идей может послужить инфраструктура TensorFlow Extended (TFX)[1]. К сфере тестирования из перечисленных в ней компонентов конвейера относятся прежде всего *средство проверки примеров данных* (example validator), *средство проверки модели* (model validator) и *средство оценки модели* (model evaluator). Схема на рис. 12.1 включает соответствующие этим компонентам прямоугольники.

Средство проверки примеров данных осуществляет тестирование данных — часто игнорируемый аспект тестирования системы машинного обучения. Знаменитое высказывание, популярное среди специалистов по машинному обучению, гласит: «Мусор на входе — мусор на выходе» (Garbage in, garbage out). Качество обученной модели машинного обучения ограничивается качеством ее входных данных. Примеры с некорректными значениями признаков или метками, вероятно, отрицательно повлияют на безошибочность обученной модели после ее развертывания (и это если сначала не возникнут проблемы при обучении модели из-за этих плохих примеров данных!). Средство проверки примеров данных обеспечивает соответствие свойств

[1] *Baylor D. et al.* TFX: A TensorFlow-Based Production-Scale Machine Learning Platform // KDD, 2017. www.kdd.org/kdd2017/papers/view/tfx-a-tensorflow-based-production-scale-machine-learning-platform.

используемых при обучении и оценке модели данных определенным требованиям: достаточности объемов данных, допустимости их распределения и отсутствию каких-либо причудливых аномальных значений. Например, рост (в сантиметрах) пациента в наборе медицинских данных не должен превышать 280; возраст пациента должен быть неотрицательным числом от 0 до 130; пероральная температура (в градусах Цельсия) должна быть положительным числом примерно между 30 и 45 и т. д. Если же какие-либо примеры данных содержат признаки, выходящие за пределы этих диапазонов, или содержат значения-«заполнители», например None или NaN, значит, с этими примерами данных что-то не в порядке, и с ними следует поступить соответствующим образом — обычно исключить из процессов обучения и оценки. Как правило, подобные ошибки указывают либо на сбой в процессе сбора данных, либо на то, что «мир изменился» несовместимым с лежащими в основе системы допущениями образом. В большинстве случаев такое тестирование скорее напоминает мониторинг, а не комплексное тестирование.

Такие компоненты, как средство проверки примеров данных, полезны также для выявления *асимметрии между обучением и выдачей результатов* (training-serving skew) — особенно неприятной разновидности программной ошибки, возникающей в системах машинного обучения. Две основные ее причины: 1) различные распределения данных, используемых при обучении и реальной работе модели, и 2) что выполнение кода идет по различным путям при обучении и реальной работе модели. Развертывание средства проверки примеров данных в обеих средах — обучения и эксплуатации модели — позволяет потенциально обнаружить ошибки, возникающие на каком-либо из этих путей выполнения кода.

Средство проверки модели играет роль человека — создателя модели, когда речь заходит о том, достаточно ли модель «хороша» для реальной эксплуатации. Достаточно настроить его, указав интересующие вас метрики качества, после чего оно «благословляет» модель или отвергает ее. Опять же, как и в случае средства проверки примеров данных, оно работает скорее в стиле мониторинга и оповещения. Обычно имеет смысл журналировать метрики качества (безошибочность и т. д.) и строить графики их зависимости от времени, чтобы вовремя выявить небольшие систематические ухудшения рабочих характеристик, которые сами по себе не вызвали бы срабатывания предупреждения, но полезны для диагностики долговременных тенденций и локализации их причин.

Средство оценки модели позволяет подробнее исследовать статистику качества работы модели, анализируя его вдоль и поперек заданной пользователем оси координат. Зачастую его применяют, чтобы «прощупать», как модель ведет себя для различных групп пользователей, в смысле возраста, образования, географии и т. д. В качестве простой иллюстрации можно привести проверку для примера ирисов из раздела 3.3, совпадает ли приблизительно безошибочность классификации для трех видов ирисов. Если контрольный или оценочный набор данных слишком смещен в сторону одной из групп, вполне возможно, что модель всегда выдает неправильные ответы для наименьшей группы, хотя проблема с безошибочностью и не проявляется слишком явно. Как и в случае со средством проверки модели, тенденции в зависимости от времени часто столь же полезны, как и измерения в отдельные моменты времени.

12.2. Оптимизация модели

После того как вы методично создали, обучили и протестировали модель, пора применить ее в деле. Этот процесс — *развертывание модели* (model deployment) — ничуть не менее важен, чем предыдущие шаги разработки модели. Неважно, отправляется ли модель на сторону клиента для выполнения вывода или будет обслуживать пользователей в прикладной части, она всегда должна работать быстро и эффективно. Если точнее, желательно, чтобы:

- размер модели был небольшим, благодаря чему она бы быстро загружалась через Интернет или с диска;
- при вызове метода `predict()` модели потреблялось как можно меньше вычислительных ресурсов, памяти и времени.

В этом разделе описываются доступные в TensorFlow.js методики для оптимизации размера и скорости выполнения вывода на основе обученных моделей перед их развертыванием.

Слово *«оптимизация»* слишком перегружено смыслами. В этом разделе мы будем понимать под *оптимизацией* усовершенствования, связанные с сокращением размеров модели и ускорения вычислений. Не путайте ее с методиками оптимизации весовых параметров в контексте обучения модели и оптимизаторов, например с градиентным спуском. Иногда их называют *качеством* (quality) и *эффективностью* (performance) модели соответственно. Эффективность отражает количество времени и ресурсов, необходимых модели для решения поставленной задачи. Качество отражает близость результатов к идеальным.

12.2.1. Оптимизация размера модели посредством квантования весовых коэффициентов модели после обучения

Веб-разработчикам совершенно очевидно, что файлы должны быть маленькими для быстрой загрузки их через Интернет. Что особенно важно, если веб-сайт ориентирован на работу с большим количеством пользователей или на пользователей с медленным интернет-соединением[1]. Кроме того, при хранении модели на мобильном устройстве (см. обсуждение мобильного развертывания с помощью TensorFlow.js в подразделе 12.3.4), размер модели часто ограничивается небольшим доступным

[1] В марте 2019 года компания Google опубликовала дудл, демонстрирующий нейронную сеть, способную сочинять музыку в стиле Иоганна Себастьяна Баха (http://mng.bz/MOQW). В основе этой браузерной нейронной сети лежал TensorFlow.js. Модель была квантована на 8-битные целые числа при помощи описанных в этом разделе методов, что сократило ее передаваемый по сети размер в несколько раз, примерно до 380 Кбайт. Без такого квантования показывать эту модель такой обширной аудитории пользователей, как у домашней страницы Google (где отображаются дудлы), было бы невозможно.

объемом памяти. Нейронные сети, отличающиеся все бóльшим и бóльшим размером, в этом смысле доставляют проблемы при развертывании модели. Высокие разрешающие возможности глубоких нейронных сетей (то есть их способности к предсказанию) часто означают и большее количество слоев и их размеры. На момент написания книги размер весовых коэффициентов передовых моделей распознавания изображений[1], распознавания речи[2], обработки текстов на естественном языке[3] и генеративных моделей[4] зачастую превышает 1 Гбайт. Вследствие непрерывного поиска баланса между величиной и возможностями моделей, оптимизация размера моделей (как спроектировать нейронную сеть как можно меньшего размера, способную тем не менее на выполнение поставленных задач со степенью безошибочности, близкой к большей нейронной сети) остается очень активной сферой исследований в глубоком обучении. Существует два основных подхода к оптимизации размера моделей. При первом подходе нейронная сеть изначально проектируется исходя из минимизации размера модели. При втором применяются методики сокращения размера уже существующих нейронных сетей.

MobileNetV2, которую мы обсуждали в посвященных сверточным сетям главах, — результат первого подхода[5]. Она представляет собой маленькую, облегченную модель для изображений, подходящую для развертывания в средах с ограниченными ресурсами, например в браузерах и мобильных устройствах. Степень безошибочности MobileNetV2 несколько ниже, чем у больших моделей, обученных на тех же задачах, например ResNet50. Однако размер ее (14 Мбайт) в несколько раз меньше (ResNet50 занимает около 100 Мбайт), так что небольшое снижение безошибочности оправдывает себя.

Несмотря на заложенное при проектировании снижение размера, MobileNetV2 все равно слишком велика для большинства JavaScript-приложений. Стоит только упомянуть, что ее размер (14 Мбайт) в восемь раз превышает размер средней веб-страницы[6]. В MobileNetV2 есть параметр `width`, при значении меньше 1 снижающий размер всех сверточных слоев, тем самым еще более сокращающий общий размер

---

1 *He K. et al.* Deep Residual Learning for Image Recognition // submitted 10 Dec. 2015. https://arxiv.org/abs/1512.03385.

2 *Schalkwyk J.* An All-Neural On-Device Speech Recognizer // Google AI Blog, 12 Mar. 2019. http://mng.bz/ad67.

3 *Devlin J. et al.* BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding // submitted 11 Oct. 2018. https://arxiv.org/abs/1810.04805.

4 *Karras T., Laine S., Aila T.* A Style-Based Generator Architecture for Generative Adversarial Networks // submitted 12 Dec. 2018. https://arxiv.org/abs/1812.04948.

5 *Sandler M. et al.* MobileNetV2: Inverted Residuals and Linear Bottlenecks // IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018. Pp. 4510–4520. http://mng.bz/NeP7.

6 Согласно данным HTTP Archive, средний размер веб-страницы (общий передаваемый размер HTML, CSS, JavaScript, изображений и прочих статических файлов) составляет примерно 1828 КБ для традиционных приложений и 1682 КБ для мобильных приложений по состоянию на май 2019 года: https://httparchive.org/reports/page-weight.

модели (и еще более снижающий степень безошибочности). Например, размер версии MobileNetV2 с `width` = 0,25 составляет лишь четверть от размера полной модели (3,5 Мбайт). Но даже это может оказаться неприемлемым для веб-сайтов с высокой посещаемостью, чувствительных к увеличению размеров страниц и времени загрузки.

Можно ли еще больше сократить размер подобных моделей? К счастью, да. Что приводит нас ко второму из упомянутых подходов, а именно, не зависящей от модели оптимизации размеров. Методики из этой категории более универсальны в том смысле, что не требуют изменений архитектуры модели, а потому применимы к широкому спектру существующих глубоких нейронных сетей. Мы сосредоточим свое внимание главным образом на методике *квантования весовых коэффициентов модели после обучения* (post-training weight quantization). Идея ее проста: сохранение, после обучения модели ее весовых коэффициентов в виде чисел с более низкой числовой точностью. Те из читателей, кому интересна математическая сторона, могут найти всю информацию в инфобоксе 12.1.

ИНФОБОКС 12.1. Математические нюансы квантования весовых коэффициентов модели после обучения

Весовые параметры нейронной сети во время обучения хранятся в виде 32-битных чисел с плавающей точкой (float32). И не только в TensorFlow.js, но и в других фреймворках глубокого обучения, например TensorFlow и PyTorch. Такое относительно ресурсоемкое представление обычно допустимо, поскольку обучение модели в большинстве случаев происходит в среде с неограниченными ресурсами (например, в среде прикладной части рабочей станции с достаточно большим объемом оперативной памяти, быстрыми CPU и GPU с поддержкой CUDA). Однако опыт показывает, что во многих сценариях выполнения вывода можно снизить точность весовых коэффициентов без существенного снижения безошибочности. Для снижения точности представления все значения типа float32 преобразуются в 8- или 16-битные целочисленные значения, отражающие дискретизированное местоположение конкретного значения в диапазоне всех значений того же весового коэффициента. Данный процесс называется *квантованием* (quantization).

В TensorFlow.js квантование весовых коэффициентов производится по отдельности. Например, если нейронная сеть состоит из четырех весовых величин (например, весов и смещений двух плотных слоев), каждый из весовых коэффициентов подвергается квантованию как одно целое. Уравнение, которое описывает квантование весового коэффициента, выглядит следующим образом:

$$\text{quantize}(w) = \text{floor}((w - w_{\text{min}}) \,/\, w_{\text{scale}} \times 2^{\text{B}}). \qquad (12.1)$$

В этом уравнении B — количество битов, выделяемое для хранения результата квантования (в настоящее время TensorFlow.js поддерживает варианты 8 и 16); w_{min} — минимальное значение параметров веса; w_{scale} — диапазон значений параметров (разница между минимумом и максимумом). Уравнение корректно, конечно, только если w_{scale}

не равен 0. В частном случае, когда w_{scale} равен 0 — то есть когда значения всех параметров веса совпадают — quantize(w) возвращает 0 для всех w.

Два вспомогательных значения, w_{min} и w_{scale}, сохраняются вместе с квантованным весовым коэффициентом для возможности восстановления весов (процесс деквантования) во время загрузки модели. Описывающее деквантование уравнение выглядит так:

$$\text{dequantize}(v) = v / 2^B \times w_{scale} + w_{min}. \quad (12.2)$$

Это уравнение справедливо вне зависимости от того, равен ли w_{scale} нулю.

Процесс квантования весовых коэффициентов модели после обучения позволяет существенно уменьшить размер модели: 16-битное квантование снижает размер модели примерно на 50 %, а 8-битное — на 75 %. Эти значения — приближенные по двум причинам. Во-первых, некоторая доля размера модели приходится на ее топологию, описываемую в JSON-файле. Во-вторых, как указано в инфобоксе выше, квантование требует хранения двух дополнительных значений с плавающей точкой (w_{min} и w_{scale}), помимо нового целочисленного значения (квантованных бит). Однако занимаемое ими место обычно незначительно по сравнению с сокращением количества битов, используемых для представления весовых параметров.

Квантование представляет собой преобразование с потерями. В результате снижения точности часть информации из исходных значений весов теряется. Что аналогично снижению битовой глубины 24-битного цветного изображения до 8-битного (подобного тем, которые можно было наблюдать на игровых приставках Nintendo в 1980-х), эффект которого хорошо заметен человеческому глазу. На рис. 12.2 приведено наглядное сравнение степени дискретизации, получающейся в результате 16- и 8-битного квантования. Как и можно было ожидать, 8-битное квантование приводит к более «крупнозернистому» представлению исходных весовых коэффициентов. При 8-битном квантовании на весь диапазон весовых параметров приходится всего 256 значений, в отличие от 65 536 значений при 16-битном квантовании. В обоих случаях снижение точности по сравнению с 32-битным представлением с плавающей точкой — разительное.

Но играет ли какую-то роль на самом деле потеря точности весовых параметров? При развертывании нейронной сети важна только степень безошибочности на контрольном наборе данных. Чтобы ответить на этот вопрос, мы скомпилируем несколько моделей, охватывающих различные задачи, в примере quantization из репозитория tfjs-examples. Для извлечения этого примера выполните:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/quantization
yarn
```

Этот пример содержит четыре сценария, каждый из которых включает свое сочетание набора данных и применяемой к нему модели. Первый сценарий связан

с предсказанием средних цен на недвижимость в различных регионах Калифорнии на основе числовых признаков наподобие медианного возраста объектов недвижимости, общего количества комнат и т. д. Модель представляет собой пятислойную сеть, включающую слои дропаута для борьбы с переобучением. Для обучения и сохранения исходной (без квантования) модели выполните команду:

```
yarn train-housing
```

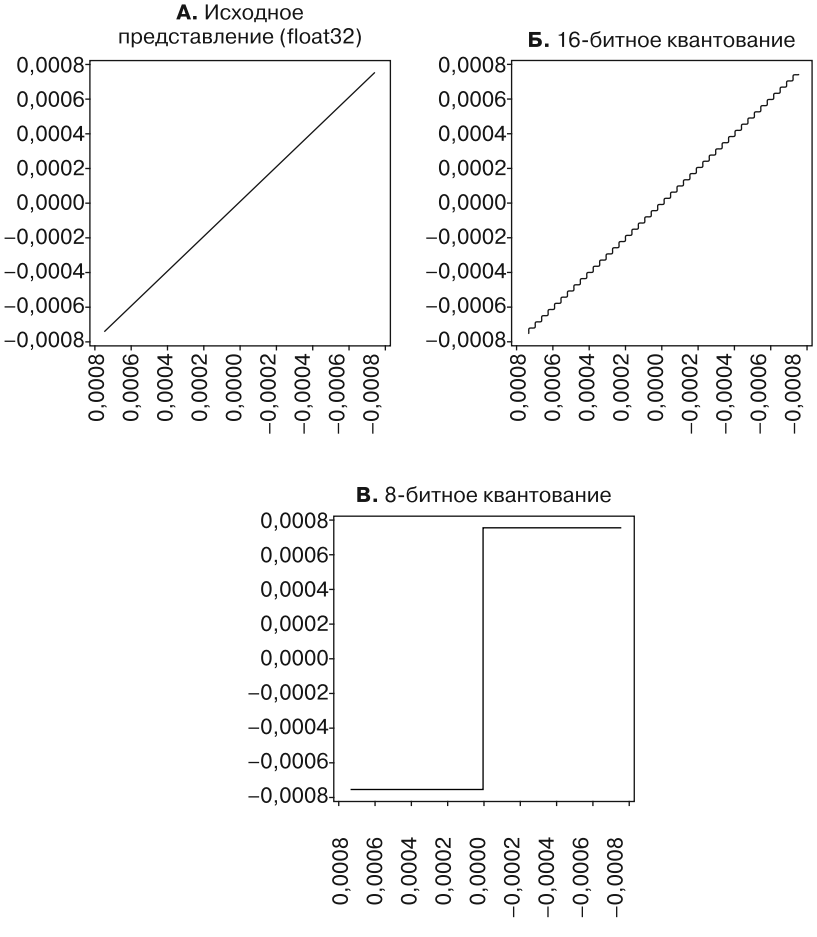


Рис. 12.2. Примеры 16- и 8-битного квантования весов. Исходная тождественная функция (y = x, блок А) уменьшается в размере при 16- и 8-битном квантовании; результаты приведены в блоках Б и В соответственно. Чтобы эффекты квантования были видны на странице книги, мы показали крупным планом небольшой участок тождественной функции в окрестности x = 0

Следующая команда выполняет 16- и 8-битное квантования сохраненной модели и оценивает, насколько эти два уровня квантования повлияли на безошибочность

модели на контрольном наборе данных (подмножестве данных, которые модель не видела во время обучения):

```
yarn quantize-and-evaluate-housing
```

Ради удобства использования в этой команде заключено множество различных операций. Впрочем, ключевой этап, на котором, собственно, производится квантование модели, виден в сценарии командной оболочки `quantization/quantize_evaluate.sh`. В этом сценарии можно видеть следующую инструкцию командной оболочки, которая производит 16-битное квантование модели, расположенной по адресу `MODEL_JSON_PATH`. Можете по образу и подобию этой команды выполнить квантование своих собственных сохраненных TensorFlow.js моделей. Если же задано значение `1` флага `--quantization_bytes`, будет произведено 8-битное квантование:

```
tensorflowjs_converter \
      --input_format tfjs_layers_model \
      --output_format tfjs_layers_model \
      --quantization_bytes 2 \
    "${MODEL_JSON_PATH}" "${MODEL_PATH_16BIT}"
```

Предыдущая команда демонстрирует, как выполнить квантование весовых коэффициентов модели, обученной на JavaScript. `tensorflowjs_converter` также поддерживает квантование весов при преобразовании моделей из Python в JavaScript, подробности которого показаны в инфобоксе 12.2.

ИНФОБОКС 12.2. Квантование весовых коэффициентов и моделей Python

В главе 5 мы показали, как модели Keras (Python) можно преобразовать в формат, подходящий для загрузки и использования в TensorFlow.js. Во время подобного преобразования Python-JavaScript можно произвести квантование весов. Для этого служит упомянутый в основном тексте флаг `--quantization_bytes`. Например, для преобразования модели в формате HDF5 (.h5), сохраненной Keras, с применением 16-битного квантования необходимо использовать следующую команду:

```
tensorflowjs_converter \
      --input_format keras \
      --output_format tfjs_layers_model \
      --quantization_bytes 2 \
      "${KERAS_MODEL_H5_PATH}" "${TFJS_MODEL_PATH}"
```

В этой команде `KERAS_MODEL_H5_PATH` — путь к экспортированной из Keras модели, а `TFJS_MODEL_PATH` — путь сохранения преобразованной модели с квантованными весовыми коэффициентами.

Конкретные показатели безошибочности могут немного отличаться при разных запусках вследствие инициализации весовых коэффициентов случайными значениями и случайной перетасовки батчей данных во время обучения. Впрочем, общие выводы остаются теми же: как демонстрирует первая строка табл. 12.1, 16-битное квантование весов ведет к незначительным изменениям в MAE предсказания цен

на недвижимость, а 8-битное квантование — к относительно значительному (но все равно крошечному в абсолютном выражении) росту MAE.

Таблица 12.1. Безошибочность при оценке четырех различных моделей при квантовании весовых коэффициентов модели после обучения

Набор данных и модель	Потери при оценке и безошибочность без квантования и при различном уровне квантования		
	32-битная точность (без квантования)	**16-битное квантование**	**8-битное квантование**
Цены на калифорнийскую недвижимость: MLP-регрессор	MAE[1] = 0,311984	MAE = 0,311983	MAE = 0,312780
MNIST: сверточная сеть	Безошибочность = = 0,9952	Безошибочность = = 0,9952	Безошибочность = = 0,9952
Fashion-MNIST: сверточная сеть	Безошибочность = = 0,922	Безошибочность = = 0,922	Безошибочность = = 0,9211
Подмножество ImageNet из 1000 изображений	Безошибочность$_1$ = = 0,618. Безошибочность$_5$ = = 0,788	Безошибочность$_1$ = = 0,624. Безошибочность$_5$ = = 0,789	Безошибочность$_1$ = = 0,280. Безошибочность$_5$ = = 0,490

Второй сценарий в примере квантования основан на хорошо знакомом нам наборе данных MNIST и архитектуре глубоких сверточных сетей. Подобно эксперименту с ценами на недвижимость, мы можем обучить исходную модель и произвести оценку на ее квантованных версиях с помощью следующих команд:

```
yarn train-mnistyarn quantize-and-evaluate-mnist
```

Как демонстрирует вторая строка табл. 12.1, ни 16-битное, ни 8-битное квантование практически не влияет на степень безошибочности модели на контрольном наборе данных. Дело в том, что наша сверточная сеть представляет собой многоклассовый классификатор, так что небольшие изменения выходных значений ее слоя могут не влиять на окончательный результат классификации, получаемый с помощью операции `argMax()`.

Можно ли считать, что это характерно для всех ориентированных на обработку изображений многоклассовых классификаторов? Даже простая сверточная сеть из этого примера достигает почти идеальной степени безошибочности. А как квантование влияет на безошибочность модели при более сложных задачах классификации изображений? Для ответа на этот вопрос рассмотрим еще два сценария в данном примере квантования.

Fashion-MNIST, с которой мы сталкивались в посвященной вариационным автокодировщикам разделе в главе 10, — более сложная задача, по сравнению с MNIST. С помощью следующих команд вы можете обучить модель на наборе

[1] Функция потерь на основе MAE используется в модели цен на калифорнийскую недвижимость. В отличие от степени безошибочности, чем ниже MAE — тем лучше.

данных Fashion-MNIST и посмотреть, как 16- и 8-битное квантование влияют на степень ее безошибочности на контрольном наборе данных:

```
yarn train-fashion-mnist
yarn quantize-and-evaluate-fashion-mnist
```

Результаты, приведенные в третьей строке табл. 12.1, указывают на небольшое снижение безошибочности на контрольном наборе данных (с 92,2 до 92,1 %) при 8-битном квантовании весов, в то время как 16-битное квантование по-прежнему никаких видимых изменений не вызывает.

Еще более сложная задача классификации изображений — задача классификации ImageNet с 1000 выходных классов. В этом случае мы скачиваем предобученную MobileNetV2 вместо обучения ее с нуля, как в трех прочих сценариях в этом примере. Предобученная модель оценивается на выборке из 1000 изображений из набора данных ImageNet в квантованном и неквантованном виде. Мы решили не оценивать на всем наборе данных ImageNet, поскольку он огромного размера (миллионы изображений), а выводы при этом особо бы не отличались.

Для оценки степени безошибочности модели в задаче ImageNet более исчерпывающим образом мы вычислили степень безошибочности как для первого, так и для первых пяти максимальных логитов. Первая представляет собой долю правильных предсказаний при учете лишь одного максимального выходного логита модели, а во втором случае предсказание считается правильным, если любой из пяти максимальных логитов включает правильную метку. Это стандартный подход при оценке степеней безошибочности модели для ImageNet, поскольку — вследствие большого числа меток классов, некоторые из которых очень близки друг к другу, — модели часто содержат правильную метку не в максимальном, а в одном из пяти максимальных логитов. Чтобы увидеть эксперимент с MobileNetV2 + ImageNet в действии, используйте команду:

```
yarn quantize-and-evaluate-MobileNetV2
```

В отличие от предыдущих трех сценариев этот эксперимент демонстрирует существенное влияние 8-битного квантования на безошибочность модели на контрольном наборе (см. четвертую строку табл. 12.1). Степень безошибочности как для первого, так и для первых пяти логитов при 8-битном квантовании MobileNet существенно ниже исходной модели, так что 8-битное квантование для оптимизации размера MobileNet не подходит. Однако безошибочность при 16-битном квантовании сравнима с неквантованной моделью[1]. Как видим, влияние квантования на безошибочность зависит от модели и данных. Для некоторых моделей и задач (например, сверточной сети MNIST) ни 16-битное, ни 8-битное квантование не приводит к существенному снижению безошибочности на контрольном наборе данных. В подобных случаях имеет смысл использовать при развертывании 8-битную модель и наслаждаться сокращением времени скачивания. Для некоторых моделей, например нашей сверточной сети Fashion-MNIST или регрессионной модели для цен на недвижимость,

[1] На самом деле в таблице можно наблюдать даже небольшое повышение степеней безошибочности, вследствие случайных колебаний на относительно небольшом контрольном наборе данных, состоящем лишь из 1000 примеров данных.

16-битное квантование никакого ощутимого снижения безошибочности не дает, а 8-битное приводит лишь к незначительному снижению безошибочности. В подобном случае нужно просто решить, перевешивают ли выгоды от дополнительного 25%-ного снижения размера модели недостатки снижения степени безошибочности. Наконец, для некоторых типов моделей (например, как в нашем случае классификации изображений ImageNet с помощью MobileNetV2) 8-битное квантование приводит к резкому падению степени безошибочности, в большинстве сценариев использования неприемлемому. Для таких задач следует применять исходную модель или 16-битное квантование.

Приведенные в примере квантования сценарии — типовые задачи, возможно несколько примитивные. Задачи, с которыми вы столкнетесь, могут оказаться более сложными и сильно отличаться от этих сценариев. Так что главный вывод: производить ли квантование модели перед развертыванием и до какой битовой глубины — эмпирические вопросы, универсального ответа на которые не существует. Необходимо экспериментировать с квантованием и тестировать полученные модели на настоящих тестовых данных, прежде чем принимать какое-либо решение. Пример 1 в конце главы позволит вам поэкспериментировать с ACGAN MNIST, обученной нами в главе 10, и решить, что лучше подходит для подобной генеративной модели — 16- или 8-битное квантование.

Квантование весовых коэффициентов и сжатие gzip

Стоит учесть также, что при 8-битном квантовании можно добиться дополнительного сокращения размера передаваемой через Интернет модели с помощью различных методик сжатия данных, например, gzip. gzip широко применяется для передачи больших файлов через Интернет. Всегда включайте сжатие gzip при передаче через Интернет файлов моделей TensorFlow.js. Неквантованные весовые коэффициенты нейронной сети с типом данных float32 обычно плохо подходят для подобного сжатия из-за шумоподобных колебаний значений параметров, содержащих мало повторяющихся паттернов. По нашим наблюдениям, gzip обычно не дает более 10–20 % сжатия для неквантованных весовых коэффициентов моделей. То же самое справедливо и для моделей с 16-битным квантованием. Однако после 8-битного квантования весов модели, коэффициент сжатия часто резко возрастает (до 30–40 % для маленьких моделей и 20–30 для больших; табл. 12.2).

Причина в малом числе интервалов значений при таком резком сокращении точности (только 256), так что многие значения (например, близкие к 0) попадают в один интервал, в результате чего число повторяющихся паттернов в бинарном представлении весовых коэффициентов сильно возрастает. Это еще одна причина для предпочтения 8-битного квантования в случаях, когда оно не приводит к неприемлемому ухудшению степени безошибочности на контрольном наборе.

В общем, благодаря квантованию весовых коэффициентов модели после обучения можно существенно сократить размер моделей TensorFlow.js, передаваемых через Интернет и сохраняемых на диске, особенно с помощью различных методик сжатия данных, например gzip. Для такого улучшения коэффициента сжатия разра-

ботчикам не требуется писать никакого кода, поскольку браузер производит разархивирование незаметно при скачивании файлов модели. Однако это не меняет объема вычислений, требуемых для вызовов, производящих вывод на основе модели. Не меняет оно и объем потребляемой CPU или GPU памяти при подобных вызовах. Дело в том, что весовые коэффициенты квантуются после загрузки (см. уравнение (12.2) в инфобоксе 12.1). Что же касается выполняемых операций и типов данных и форм выходных тензоров этих операций, то никакой разницы между квантованной и неквантованной моделью нет. Однако не менее важно для развертывания модели, чтобы она работала как можно быстрее и потребляла как можно меньше памяти ради улучшения впечатлений пользователей от ее использования и снижения потребления электроэнергии. Можно ли добиться ускорения работы существующей модели TensorFlow.js при развертывании, без снижения безошибочности предсказаний вдобавок к оптимизации размера модели? К счастью, можно. В следующем разделе мы сосредоточим свое внимание на существующих в TensorFlow.js методиках оптимизации скорости выполнения вывода.

Таблица 12.2. Коэффициенты сжатия артефактов модели с помощью gzip при различном уровне квантования

Набор данных и модель	**Коэффициент сжатия с помощью gzip**[1]		
	32-битная точность (без квантования)	**16-битное квантование**	**8-битное квантование**
Цены на калифорнийскую недвижимость: MLP-регрессор	1,121	1,161	1,388
MNIST: сверточная сеть	1,082	1,037	1,184
Fashion-MNIST: сверточная сеть	1,078	1,048	1,229
Подмножество ImageNet из 1000 изображений	1,085	1,063	1,271

12.2.2. Оптимизация скорости выполнения вывода с помощью преобразования GraphModel

Этот раздел организован следующим образом. Сначала мы опишем этапы оптимизации скорости выполнения вывода модели TensorFlow.js с помощью преобразования `GraphModel`. А затем перечислим подробно показатели быстродействия, количественно выражающие прирост скорости благодаря этому подходу. И наконец, мы расскажем, что происходит «под капотом» преобразования `GraphModel`.

[1] Общий размер model.json и файла весовых коэффициентов/размер сжатого файла в формате gzip.

Пусть дана модель TensorFlow.js, сохраненная по пути `my/layers-model`; преобразовать ее в объект `tf.GraphModel` можно с помощью следующей команды:

```
tensorflowjs_converter \
      --input_format tfjs_layers_model \
      --output_format tfjs_graph_model \
      my/layers-model my/graph-model
```

Эта команда создает файл `model.json` в выходном каталоге `my/graph-model` (если данный каталог не существует, он будет создан), вместе с несколькими бинарными файлами для весовых коэффициентов. На первый взгляд может показаться, что формат этого набора файлов идентичен файлам из входного каталога, содержащего сериализованный объект `tf.LayersModel`. Однако выходные файлы кодируют иную разновидность модель — `tf.GraphModel` (по которой данный метод оптимизации и назван). Для загрузки преобразованной модели в браузере или Node.js необходимо использовать метод `tf.loadGraphModel()` вместо привычного вам `tf.loadLayersModel()`. После загрузки объекта `tf.GraphModel` можно выполнять вывод совершенно так же, как и в случае `tf.LayersModel`, с помощью вызова метода `predict()` этого объекта. Например:

```
const model = await tf.loadGraphModel('file://./my/graph-model/model.json');
const ys = model.predict(xs);
```

Либо можно указать URL вида http:// или https:// при загрузке модели в браузере

Выполнение вывода на основе входных данных 'xs'

Увеличение скорости вывода сопряжено с двумя ограничениями.

- На момент написания данной книги последняя версия TensorFlow.js (1.1.2) не поддерживает преобразования в `GraphModel` рекуррентных слоев, таких как `tf.layers.simpleRNN()`, `tf.layers.gru()` и `tf.layers.lstm()` (см. главу 9).
- У загруженного объекта `tf.GraphModel` отсутствует метод `fit()`, так что возможности дальнейшего обучения (например, переноса обучения) нет.

В табл. 12.3 приведено сравнение скоростей выполнения вывода для двух типов моделей с преобразованием `GraphModel` и без него. Поскольку преобразование в `GraphModel` рекуррентных слоев пока что не поддерживается, приведены только результаты для MLP и сверточной сети (MobileNetV2). Чтобы охватить развертывание в различных средах, в таблице представлены результаты как для браузера, так и для tfjs-node, запущенного в среде прикладной части. Из этой таблицы видно, что преобразование `GraphModel` всегда ускоряет выполнение вывода, однако коэффициент ускорения зависит от типа модели и среды развертывания. Для браузерной среды развертывания (WebGL), преобразование `GraphModel` дает ускорение на 20–30 %, в то время как при развертывании в среде Node.js ускорение намного более впечатляющее (70–90 %). Далее мы обсудим, почему преобразование `GraphModel` ускоряет вывод, а также почему ускорение больше в случае Node.js, чем в браузерной среде.

Таблица 12.3. Сравнение скорости выполнения вывода двух типов моделей (MLP и MobileNetV2) с оптимизацией и без нее в виде преобразования GraphModel в различных средах развертывания[1]

Название и топология модели	Время выполнения predict() (мс; чем меньше, тем лучше) (Усредненное значение, по 30 вызовам predict(), с 20 предшествующими вызовами, «для прогрева»)					
	WebGL в браузере		tfjs-node (только CPU)		tfjs-node-gpu	
	LayersModel	GraphModel	LayersModel	GraphModel	LayersModel	GraphModel
MLP[2]	13	10 (1.3x)	18	10 (1.8x)	3	1.6 (1.9x)
MobileNetV2 (width = 1.0)	68	57 (1.2x)	187	111 (1.7x)	66	39 (1.7x)

Как преобразование GraphModel ускоряет выполнение вывода на основе модели

Как преобразование `GraphModel` повышает скорость выполнения вывода моделей TensorFlow.js? Благодаря упреждающему анализу графа вычислений модели TensorFlow (Python) на уровне мелких структурных единиц. После анализа графа вычислений он модифицируется таким образом, чтобы снизить объем необходимых вычислений с сохранением численной правильности выходных результатов графа. Не пугайтесь терминов *«упреждающий анализ»* (ahead-of-time analysis) и *«на уровне мелких структурных единиц»* (fine granularity). Дальше мы их немного поясним.

В качестве конкретного примера подобной модификации графа вычислений рассмотрим функционирование слоя BatchNormalization в `tf.LayersModel` и `tf.GraphModel`. Напомним, что BatchNormalization — слой, улучшающий сходимость и снижающий переобучение во время обучения. Он доступен в API TensorFlow.js в виде метода `tf.layers.batchNormalization()` и используется во многих популярных предобученных моделях, например в MobileNetV2. При использовании этого слоя в качестве части `tf.LayersModel` производимые вычисления следуют математическому определению нормализации по батчам:

$$\text{выходной\_сигнал} = (x - \text{mean}) / (\text{sqrt}(\text{var}) + \text{epsilon}) \times \text{gamma} + \text{beta}. \quad (12.3)$$

Для генерации выходного сигнала на основе входного (`x`) требуется вычислить шесть операций приблизительно в следующем порядке.

1. `sqrt`, с `var` в качестве входного значения.
2. `add` с `epsilon` и результатом первого шага в качестве входных значений.

---

[1] Код, с помощью которого были получены эти результаты, можно найти по адресу https://github.com/tensorflow/tfjs/tree/master/tfjs/integration_tests/.

[2] Этот многослойный перцептрон состоит из плотных слоев со следующим количеством нейронов: 4000, 1000, 5000 и 1. У первых трех из этих слоев функция активации ReLU; у последнего — линейная.

3. `sub` с `x` и `means` в качестве входных значений.
4. `div` с результатами шагов 2 и 3 в качестве входных значений.
5. `mul` с `gamma` и результатом шага 4 в качестве входных значений.
6. `add` с `beta` и результатом шага 5 в качестве входных значений.

На основе очевидных арифметических правил можно существенно упростить уравнение (12.3), если значения `mean`, `var`, `epsilon`, `gamma` и `beta` постоянны (не меняются в зависимости от входных данных или количества вызовов слоя). После обучения включающей слой BatchNormalization модели все эти переменные действительно становятся константами. Именно так и работает преобразование `GraphModel` — схлопывает константы и упрощает арифметику, в результате чего получается следующее математически эквивалентное уравнение:

$$\text{output} = x * k + b. \tag{12.4}$$

Значения k и b вычисляются во время преобразования `GraphModel`, а не во время выполнения вывода:

$$k = \text{gamma} / (\text{sqrt}(\text{var}) + \text{epsilon}); \tag{12.5}$$

$$b = -\text{mean} / (\text{sqrt}(\text{var}) + \text{epsilon}) \times \text{gamma} + \text{beta}. \tag{12.6}$$

Следовательно, на объем вычислений во время выполнения вывода *не* влияют уравнения (12.5) и (12.6), а только уравнение (12.4). При сравнении уравнений (12.3) и (12.4) видно, что схлопывание констант и упрощение арифметики сокращает количество операций с шести до двух (операции `mul` между x и k и операции `add` между b и результатом операции `mul`), что значительно ускоряет выполнение данного слоя. Но почему `tf.LayersModel` не производит подобной оптимизации? Из-за необходимости поддержки обучения слоя BatchNormalization, во время которого значения `mean`, `var`, `gamma` и `beta` обновляются на каждом шаге обучения. Преобразование `GraphModel` извлекает выгоду из того факта, что эти обновленные значения более не нужны по завершении обучения модели.

Оптимизация, подобная показанной в примере BatchNormalization, возможна лишь при соблюдении двух требований. Во-первых, вычисления должны производиться на уровне достаточно мелких структурных единиц, то есть на уровне элементарных математических операций наподобие `mul` и `add`, а не более крупных послойных операций, на которых основан API слоев TensorFlow.js. Во-вторых, все необходимые операции должны быть известны заранее, до вызовов метода `predict()` модели. Преобразование `GraphModel` производится через библиотеку TensorFlow (Python), у которой есть доступ к графовому представлению модели, удовлетворяющему обоим критериям.

Помимо обсуждавшихся выше схлопывания констант и арифметической оптимизации, преобразование `GraphModel` способно еще на один вид оптимизации — *слияние операций* (op fusion). Для примера возьмем часто используемый тип плотного слоя (`tf.layers.dense()`). Плотный слой включает три операции: матричное умножение (`matMul`) входного сигнала x и ядра W, сложение с транслированием результата

`matMul` и смещения (b), а также поэлементная функция активации ReLU (рис. 12.3, блок А). Слияние операций состоит в замене этих трех отдельных операций одной, производящей все эквивалентные им шаги (см. рис. 12.3, блок Б). Подобная замена может показаться тривиальной, однако приводит к ускорению вычислений, благодаря: 1) уменьшению накладных расходов на выполнение операций (да, выполнение операции всегда требует определенного количества накладных расходов, вне зависимости от используемой прикладной части) и 2) расширению возможностей использования различных приемов оптимизации скорости внутри реализации самой объединенной операции.

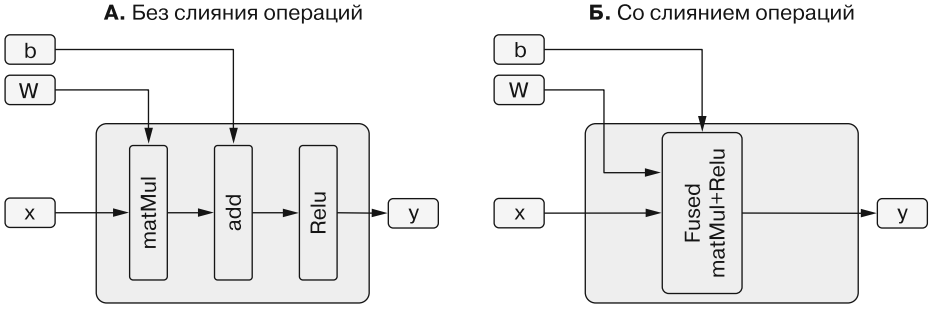


Рис. 12.3. Схематическая иллюстрация внутренних операций плотного слоя, со слиянием операций (блок А) и без него (блок Б)

Чем слияние операций отличается от вышеупомянутого схлопывания констант и арифметических упрощений? Слияние операций требует описания специальной объединенной операции (`Fused matMul+relu` в данном случае) и ее доступности для осуществляющей вычисления прикладной части, в то время как схлопывание констант — нет. Эти специальные объединенные операции доступны только в некоторых прикладных частях и средах развертывания. Именно поэтому выполнение вывода ускоряется намного сильнее в среде Node.js, чем в браузере (см. табл. 12.3). Вычислительная прикладная часть Node.js, использующая написанную на C++ и CUDA библиотеку libtensorflow, включает намного больший набор операций, чем прикладная часть WebGL TensorFlow.js в браузере.

Помимо схлопывания констант, арифметических упрощений и слияния операций, система Grappler оптимизации графов TensorFlow (Python) умеет производить еще несколько видов оптимизаций, часть из которых уместно упомянуть в свете того, как оптимизируются модели TensorFlow.js с помощью преобразования `GraphModel`. Однако объем нашей книги ограничен, так что мы не будем описывать их тут. Если вам интересно узнать больше по этому вопросу, рекомендуем взглянуть на познавательные презентации Расмуса Ларсена и Татьяны Шпейзман, указанные в конце главы.

Подытоживаем: преобразование `GraphModel` — возможность, предоставляемая командой `tensorflowjs_converter`, которая использует возможности упреждающей оптимизации графов TensorFlow (Python) для упрощения графов вычислений

и сокращения требуемых для выполнения вывода вычислений. Хотя степень ускорения выполнения вывода различается в зависимости от типа модели и используемой прикладной части, обычно она составляет 20 % или более, а потому рекомендуем вам применять это преобразование к своим моделям TensorFlow.js перед их развертыванием.

ИНФОБОКС 12.3. Правильная оценка времени выполнения вывода на основе моделей TensorFlow.js

Как объект `tf.LayersModel`, так и `tf.GraphModel` предоставляют единый метод `predict()` для выполнения вывода, принимающий на входе один или несколько тензоров и возвращающий один или несколько тензоров в качестве результата вывода. Однако важно отметить в контексте вывода на основе WebGL в браузере, что метод `predict()` лишь *планирует* операции на выполнение в GPU, а не ожидает завершения их выполнения. В результате, если «наивно» попытаться оценить длительность выполнения вызова `predict()` следующим образом, результат окажется неправильным:

```
console.time('TFjs inference');
const outputTensor = model.predict(inputTensor);
console.timeEnd('TFjs inference');
```

Некорректный способ измерения времени выполнения вывода!

Возврат из метода `predict()` не означает, что выполнение запланированных операций завершено. Следовательно, предыдущий пример приведет к заниженной оценке времени, фактически затраченного на выполнение вывода. Чтобы убедиться, что операции завершены, необходимо, прежде чем вызывать `console.timeEnd()`, выполнить один из следующих методов возвращаемого объекта тензора: `array()` или `data()`. Оба метода скачивают из GPU в CPU значения текстур, содержащих элементы выходного тензора, для чего им необходимо дождаться завершения вычисления выходного тензора. Итак, правильный способ оценки времени вывода выглядит следующим образом:

```
console.time('TFjs inference');
const outputTensor = model.predict(inputTensor);
await outputTensor.array();
console.timeEnd('TFjs inference');
```

Возврат из вызова array() не произойдет до тех пор, пока не будут завершены запланированные вычисления outputTensor, так что правильность измерения времени выполнения вывода гарантирована

Следует учесть еще один важный нюанс: как и в случае всех прочих JavaScript-программ, время выполнения вывода на основе модели TensorFlow.js — величина непостоянная. Для получения надежной оценки выполнения вывода необходимо поместить код из предыдущего фрагмента в цикл `for` и произвести измерения несколько раз (например, 50), после чего вычислить среднее значение накопленных отдельных измерений. Первые несколько раз выполнение происходит медленнее, чем далее, из-за необходимости компиляции новых шейдеров WebGL и задания начальных значений. Поэтому в коде для оценки быстродействия обычно пропускается несколько (например, пять) первых проходов, называемых *прогревочными* (burn-in, warm-up).

Чтобы лучше разобраться в подобных методиках измерения быстродействия, выполните упражнение 3 в конце данной главы.

12.3. Развертывание моделей TensorFlow.js на различных платформах и в различных средах

Вы оптимизировали модель, она быстрая и облегченная, и все тесты пройдены. Поехали! Ура! Но прежде, чем открывать шампанское, нужно еще кое-что сделать.

Пришло время включить модель в приложение и показать ее вашим пользователям. В этом разделе мы обсудим несколько платформ для развертывания. Наиболее известные варианты: развертывание в веб-платформе и развертывание на платформе сервиса Node.js, но мы рассмотрим и несколько более экзотических сценариев наподобие развертывания на основе расширения браузера или приложения одноплатного встроенного компьютера. Мы приведем простые примеры и обсудим особые нюансы каждой из этих платформ.

12.3.1. Дополнительные нюансы развертывания на веб-платформе

Начнем с того, что обратимся снова к наиболее распространенному для моделей TensorFlow.js сценарию — развертыванию на веб-платформе в виде части веб-страницы. При этом сценарии обученная, а возможно, и оптимизированная модель загружается через JavaScript с какого-либо сервера, после чего производит предсказания с помощью движка JavaScript из браузера пользователя. Хороший пример этого сценария — пример классификации изображений с помощью MobileNet, из главы 5. Можете загрузить этот пример из каталога `tfjs-examples/mobilenet`. Напомним, что соответствующий код загрузки модели и выполнения вывода выглядит вот так:

```
const MOBILENET_MODEL_PATH =
    'https://storage.googleapis.com/tfjs-
     models/tfjs/mobilenet_v1_0.25_224/model.json';
const mobilenet = await tf.loadLayersModel(MOBILENET_MODEL_PATH);
const response = mobilenet.predict(userQueryAsTensor);
```

Модель загружается из бакета платформы Google Cloud (Google Cloud Platform, GCP). В случае подобных статических приложений с небольшим объемом трафика можно легко разместить модель статически вместе с прочим содержимым сайта. В более крупных, с большим объемом трафика приложениях имеет смысл разместить модель в сети доставки контента (content delivery network, CDN) вместе с прочими тяжеловесными ресурсами. Не забывайте учитывать совместное использование ресурсов разными источниками (Cross-Origin Resource Sharing, CORS) при настройке бакетов GCP, Amazon S3 или других облачных сервисов — это распространенная ошибка при разработке. При неправильной настройке CORS модель не загрузится и в консоль будет выведено связанное с CORS сообщение об ошибке. Стоит обратить внимание на этот нюанс, если ваше приложение прекрасно работает на локальной машине, но отказывает после развертывания на платформе эксплуатации.

После загрузки HTML- и JavaScript-ресурсов браузером пользователя интерпретатор JavaScript выполняет вызов для загрузки модели. Процесс загрузки маленькой модели в современном браузере при хорошем интернет-соединении занимает несколько сот миллисекунд, но после первоначальной загрузки она будет загружаться намного быстрее из кэша браузера. Формат сериализации гарантирует сегментирование модели на фрагменты, достаточно малые для того, чтобы поместиться в кэш стандартного браузера.

Одно из преимуществ развертывания на веб-платформе — предсказание производится непосредственно в браузере. Никакие передаваемые модели данные не пересылаются через Интернет, что сокращает задержку и обеспечивает защиту персональной информации. Представьте себе сценарий предсказания текстового ввода, при котором модель предсказывает следующее слово для упрощения набора текста, нечто регулярно встречающееся, например, в Gmail. Если отправлять набранный текст на сервер в облаке и ждать ответа этих удаленных серверов, то предсказание будет выдаваться с задержкой и пользы от таких предсказаний будет намного меньше. Более того, некоторые пользователи считают, что отправка наполовину набранного ими текста на удаленный компьютер — посягательство на их персональную информацию. Выполнение предсказаний в локальном браузере — намного более безопасный способ.

Недостаток выполнения предсказаний в браузере — безопасность модели. Если модель отправляется пользователю, он может с легкостью сохранить ее и воспользоваться ею для других целей. В TensorFlow.js на момент написания данной книги (2019 год) нет решения проблемы безопасности модели в браузере. Некоторые другие варианты развертывания затрудняют использование модели не по назначению. Чтобы добиться максимальной безопасности модели, держите модель на подконтрольных вам серверах и обслуживайте запросы на предсказание там. Конечно, за счет дополнительной задержки и ухудшения защиты персональной информации.

12.3.2. Развертывание в облачных сервисах

Многие из существующих промышленных систем предоставляют предсказания на основе машинного обучения в качестве сервиса, например, Google Cloud Vision AI (https://cloud.google.com/vision) и Microsoft Cognitive Services (https://azure.microsoft.com/en-us/services/cognitive-services). Конечный пользователь подобного сервиса выполняет HTTP-запросы к нему, включающие входные данные для предсказания, например изображение для задачи обнаружения объектов, а в получаемом ответе кодируется предсказание, например метки и расположение объектов в изображении.

По состоянию на 2019 год есть два способа выдачи результатов модели TensorFlow.js с сервера. Первый — использовать сервер с Node.js и выполнять предсказание с помощью нативной среды выполнения JavaScript. В силу новизны TensorFlow.js нам неизвестны промышленные сценарии использования на основе этого подхода, однако создать работоспособный прототип для него несложно.

Второй способ: преобразовать модель из TensorFlow.js в формат, допускающий выдачу с помощью известной существующей серверной технологии, например стандартной системы TensorFlow Serving. Цитата из документации по адресу http://www.tensorflow.org/tfx/guide/serving: *«TensorFlow Serving — гибкая и высокопроизводительная система выдачи для моделей машинного обучения, рассчитанная на среду промышленной эксплуатации. TensorFlow Serving упрощает развертывание новых алгоритмов и экспериментов с сохранением архитектуры и API сервера. Система TensorFlow Serving интегрируется "из коробки" с моделями TensorFlow, но ее легко можно расширить на другие типы моделей и данных»*.

Пока что мы сериализовали все модели TensorFlow.js только в JavaScript-форматы. Система TensorFlow Serving ожидает модели в стандартном формате TensorFlow — SavedModel. К счастью, благодаря проекту tfjs-converter преобразование в требуемый формат не представляет трудностей.

В главе 5 (посвященной переносу обучения) мы показали, как использовать модели в формате SavedModel, созданные с помощью Python-реализации TensorFlow, в TensorFlow.js. Для обратного сначала необходимо установить пакет tensorflowjs из системы управления пакетами pip:

```
pip install tensorflowjs
```

Далее необходимо запустить исполняемый файл, указав входные данные:

```
tensorflowjs_converter \
    --input_format=tfjs_layers_model \
    --output_format=keras_saved_model \
    /path/to/your/js/model.json \
    /path/to/your/new/saved-model
```

В результате выполнения этой команды создается новый каталог `saved-model` с требуемой топологией и весовыми коэффициентами в понятном TensorFlow Serving формате. После этого вы можете следовать инструкциям по созданию сервера TensorFlow Serving и выполнять к работающей модели gRPC-запросы на предсказание. Существуют и управляемые решения. Например, Google Cloud Machine Learning Engine позволяет загружать сохраненные модели в Cloud Storage и выполнять их как сервис, без необходимости поддержания работы сервера или отдельной машины. Дополнительную информацию об этом можно найти по адресу https://cloud.google.com/ml-engine/docs/tensorflow/deploying-models.

Преимущество работы модели в облачном сервисе — полный контроль над ней: удобная телеметрия выполняемых запросов и быстрота обнаружения проблем. При обнаружении непредвиденной проблемы в модели можно быстро ее отключить или обновить без риска наличия ее копий на неподконтрольных вам машинах. Недостатки: уже упоминавшиеся выше дополнительная задержка и проблемы с защитой персональной информации. Работа с облачным сервисом означает также дополнительные затраты — как в денежном выражении, так и в смысле затрат труда на сопровождение, поскольку за конфигурацию системы отвечаете вы.

12.3.3. Развертывание в среде браузерного расширения, например Chrome Extension

Для некоторых работающих на стороне клиента приложений необходимо, чтобы ваше приложение могло работать на множестве различных веб-сайтов. Фреймворки создания браузерных расширений существуют для всех основных браузеров для настольных компьютеров, в том числе Chrome, Safari и FireFox, помимо прочих. С помощью этих фреймворков разработчики могут создавать расширения, меняющие или улучшающие впечатления пользователя от работы с браузером за счет добавления новых сценариев JavaScript или манипуляций с DOM сайтов.

А поскольку браузерные расширения работают поверх JavaScript и HTML в механизме выполнения браузера, с TensorFlow.js в браузерном расширении можно делать то же, что и при обычном развертывании на веб-странице. Безопасность модели и защита персональных данных такие же, как и при развертывании на веб-странице. Благодаря выполнению предсказания непосредственно в браузере данные пользователей находятся в относительной безопасности. Безопасность модели также аналогична таковой при развертывании на веб-странице.

В качестве иллюстрации возможностей браузерного расширения рассмотрим пример chrome-extension из репозитория tfjs-examples. Это расширение загружает модель MobileNetV2 и применяет ее к выбранным пользователям изображениям из Интернета. Установка и использование этого примера несколько отличаются от предыдущих примеров, поскольку речь идет о браузерном расширении, а не хостируемом сайте. Для этого примера необходим браузер Chrome[1].

Во-первых, необходимо скачать данное расширение и произвести его сборку аналогично прочим примерам:

```
git clone https://github.com/tensorflow/tfjs-examples.git
cd tfjs-examples/chrome-extension
yarn
yarn build
```

После сборки расширения можно загрузить распакованное расширение в Chrome. Для этого необходимо перейти по адресу chrome://extensions, включить Developer mode (Режим разработчика) и нажать Load Unpacked (Загрузить распакованные), как показано на рис. 12.4, что приведет к отображению диалогового окна выбора файла, в котором нужно выбрать созданный внутри каталога `chrome-extension` подкаталог `dist`, содержащий `manifest.json`.

После установки расширения можно приступить к классификации изображений в браузере. Для этого перейдите на какой-либо сайт с изображениями, например на страницу поиска изображений Google для ключевого слова *tiger*. Затем щелкните правой кнопкой мыши на изображении, которое хотели бы классифицировать. Вы увидите пункт меню Classify Image with TensorFlow.js (Классифицировать изображение с помощью TensorFlow.js). Выбор этого пункта меню приводит к тому, что расширение запускает для данного изображения модель MobileNet, после чего накладывает поверх изображения соответствующий предсказанию текст (рис. 12.5).

[1] Последние версии Microsoft Edge также ограниченно поддерживают загрузку совместимых с несколькими браузерами расширений.

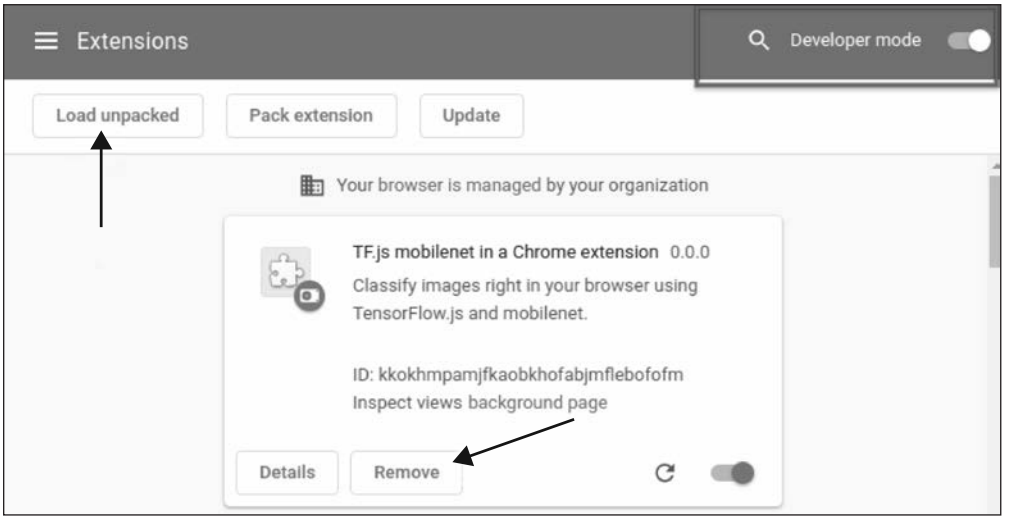


Рис. 12.4. Загрузка расширения браузера Chrome для модели MobileNet TensorFlow.js в режиме разработчика

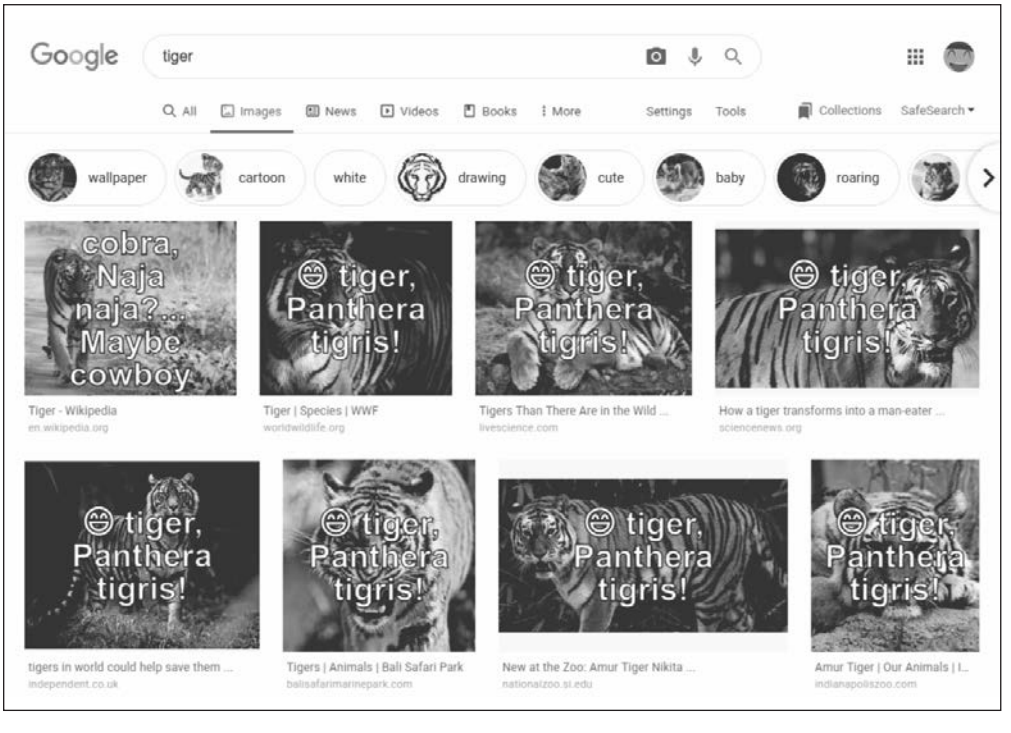


Рис. 12.5. Классификация изображений с помощью расширения браузера Chrome для модели MobileNet TensorFlow.js

Для удаления данного расширения нажмите Remove (Удалить) на странице Extensions (см. рис. 12.4) или воспользуйтесь пунктом Remove меню, выводимого Chrome при щелчке правой кнопкой мыши на значке расширения справа вверху.

Отметим, что выполняемая в браузерном расширении модель может пользоваться теми же возможностями аппаратного ускорения, что и модель, работающая на веб-странице, и, конечно, их код в значительной мере совпадает. Загружается модель с помощью вызова `tf.loadGraphModel(...)` с указанием соответствующего URL, а предсказания выполняются с помощью уже знакомого нам API `model.predict(...)`. Миграция технологии (создание прототипа) с версии для веб-страницы в браузерное расширение относительно проста.

12.3.4. Развертывание моделей TensorFlow.js в мобильных JavaScript-приложениях

Для многих программных продуктов браузеры для настольных компьютеров не охватывают достаточную территорию, а мобильные браузеры не обеспечивают ожидаемых пользователями возможностей настройки продукта под свои нужды с плавной анимацией. Перед работающими над подобными проектами разработчиками часто встает проблема организации базы кода для веб-приложений параллельно с репозиториями кода нативных приложений для операционной системы Android (на языках Java или Kotlin) и iOS (на языках Objective C или Swift). И хотя очень крупные организации могут позволить себе подобные расходы, все больше разработчиков решает переиспользовать значительную часть кода для этих вариантов развертывания за счет применения фреймворков разработки межплатформенных приложений.

Благодаря фреймворкам разработки межплатформенных приложений, таким как React Native, Ionic, Flutter и Progressive WebApps, можно написать основную часть приложения один раз на едином языке, скомпилировать эту базовую функциональность и получить на каждой платформе ожидаемые пользователями нативные вид, ощущения и быстродействие. Язык/среда выполнения берет на себя большую часть бизнес-логики и отображения UI и подключается к привязкам нативной платформы для получения стандартного для нее внешнего вида и ощущений пользователя. Выбор подходящего фреймворка разработки гибридных приложений — тема бесчисленных блогов и видео, которые можно найти в Интернете, так что мы не станем останавливаться на этом здесь, а сосредоточим свое внимание только на одном популярном фреймворке — React Native. На рис. 12.6 показано минимальное приложение React Native, включающее модель MobileNet. Обратите внимание на отсутствие верхней полоски браузера. Хотя в этом простом приложении нет каких-либо элементов UI, если бы они были, то полностью соответствовали бы виду и ощущениям от нативных приложений Android. Аналогичное справедливо и для приложения, собранного для iOS.

К счастью, среда выполнения JavaScript в React Native поддерживает TensorFlow.js нативным образом, без каких-либо дополнительных действий. Пакет tfjs-react-native

все еще находится на этапе альфа-версии[1], но уже поддерживает GPU с помощью WebGL и пакета expo-gl. Пользовательский код выглядит примерно вот так (листинг 12.2):

```
import * as tf from '@tensorflow/tfjs';
import '@tensorflow/tfjs-react-native';
```

Данный пакет также предоставляет специальный API для загрузки и сохранения ресурсов моделей изнутри мобильного приложения.

Хотя для разработки нативных приложений с помощью React Native необходимо изучить несколько новых инструментов, в частности Android Studio для Android и XCode для iOS, кривая обучения все равно более пологая, чем если сразу углубиться в нативную разработку. Поддержка TensorFlow.js этими фреймворками разработки гибридных приложений дает возможность использовать единую базу кода для логики машинного обучения, а не разрабатывать, поддерживать и тестировать отдельные версии для каждой разновидности аппаратного обеспечения — сплошные преимущества для разработчиков, желающих предоставить пользователям ощущение работы с нативными приложениями! Но как насчет ощущения работы с нативными приложениями для настольных компьютеров?

Рис. 12.6. Снимок экрана примера нативного приложения Android, созданного с помощью React Native. В данном случае модель MobileNet TensorFlow.js выполняется внутри нативного приложения

Листинг 12.2. Загрузка и сохранение модели внутри мобильного приложения с помощью React Native

```
import * as tf from '@tensorflow/tfjs';
import {asyncStorageIO} from '@tensorflow/tfjs-react-native';

async trainSaveAndLoad() {
   const model = await train();
   await model.save(asyncStorageIO(
       'custom-model-test'))
   model.predict(tf.tensor2d([5], [1, 1])).print();
   const loadedModel =
     await tf.loadLayersModel(asyncStorageIO(
         'custom-model-test'));
   loadedModel.predict(tf.tensor2d([5], [1, 1])).print();
 }
```

Сохраняем модель в AsyncStorage — простую глобальную систему хранения приложения типа «ключ/значение»

Загружаем модель из AsyncStorage

[1] В настоящее время он уже достиг стабильной версии 0.5.0 (по состоянию на февраль 2021 года). — *Примеч. пер.*

12.3.5. Развертывание моделей TensorFlow.js в межплатформенных приложениях для настольных компьютеров на JavaScript

JavaScript-фреймворки, такие как Electron.js, позволяют писать межплатформенные приложения для настольных компьютеров в чем-то аналогично написанию межплатформенных мобильных приложений на React Native. При использовании подобных фреймворков достаточно написать код один раз, чтобы потом развертывать и выполнять его на всех основных операционных системах для настольных компьютеров, включая macOS, Windows и основные дистрибутивы Linux. Это существенное упрощение по сравнению с традиционным технологическим процессом разработки, с отдельными базами кода для по большей части несовместимых операционных систем для настольных компьютеров. Возьмем для примера Electron.js, ведущий фреймворк из этой категории. В нем в качестве виртуальной машины, служащей базой для основного процесса приложения, используется Node.js; для GUI приложения используется Chromium — полноценный, хотя и облегченный браузер, значительная часть кода которого едина с Google Chrome.

TensorFlow.js совместим с Electron.js, как демонстрирует простой пример из репозитория tfjs-examples. Этот пример, который вы можете найти в каталоге electron, демонстрирует развертывание модели TensorFlow.js для выполнения вывода в основанном на Electron.js приложении для настольных компьютеров, предназначенном для поиска файлов изображений в файловой системе, соответствующих визуально одному или нескольким ключевым словам (рис. 12.7). Этот процесс поиска включает применение модели MobileNet TensorFlow.js к содержащему изображения каталогу для выполнения вывода.

Несмотря на простоту, это приложение иллюстрирует важный нюанс развертывания моделей TensorFlow.js в приложениях Electron.js — выбор осуществляющей вычисления прикладной части. Приложение Electron.js может работать как в прикладной части на основе Node.js, так и в процессе клиентской части на основе Chromium. TensorFlow.js может работать в любой из этих сред. В результате одна и та же модель может работать как в процессе прикладной части наподобие node, так и в процессе клиентской части наподобие браузера. В случае развертывания в прикладной части используется пакет @tensorflow/tfjs-node, а при развертывании в клиентской части — @tensorflow/tfjs (рис. 12.8). С помощью кнопки-флажка в GUI нашего примера приложения можно переключаться между режимами выполнения вывода в прикладной/клиентской части (рис. 12.7), хотя в настоящем приложении на основе TensorFlow.js и Electron.js, среда, в которой работает модель, обычно выбирается заранее. Далее мы вкратце обсудим за и против этих вариантов.

Как демонстрирует рис. 12.8, при выборе различных вариантов прикладной части вычисления глубокого обучения осуществляются на различном аппаратном обеспечении. При развертывании на основе пакета @tensorflow/tfjs-node рабочая нагрузка ложится на CPU, при участии многопоточной, поддерживающей SIMD библиотеки libtensorflow. Благодаря отсутствию в среде прикладной части ограни-

чений на ресурсы развертывание моделей на основе Node.js позволяет достичь более высокой скорости работы и использовать модели большего размера. Однако основной недостаток — большой размер пакетов вследствие большого размера библиотеки libtensorflow (для tfjs-node — примерно 50 Мбайт в сжатом виде).

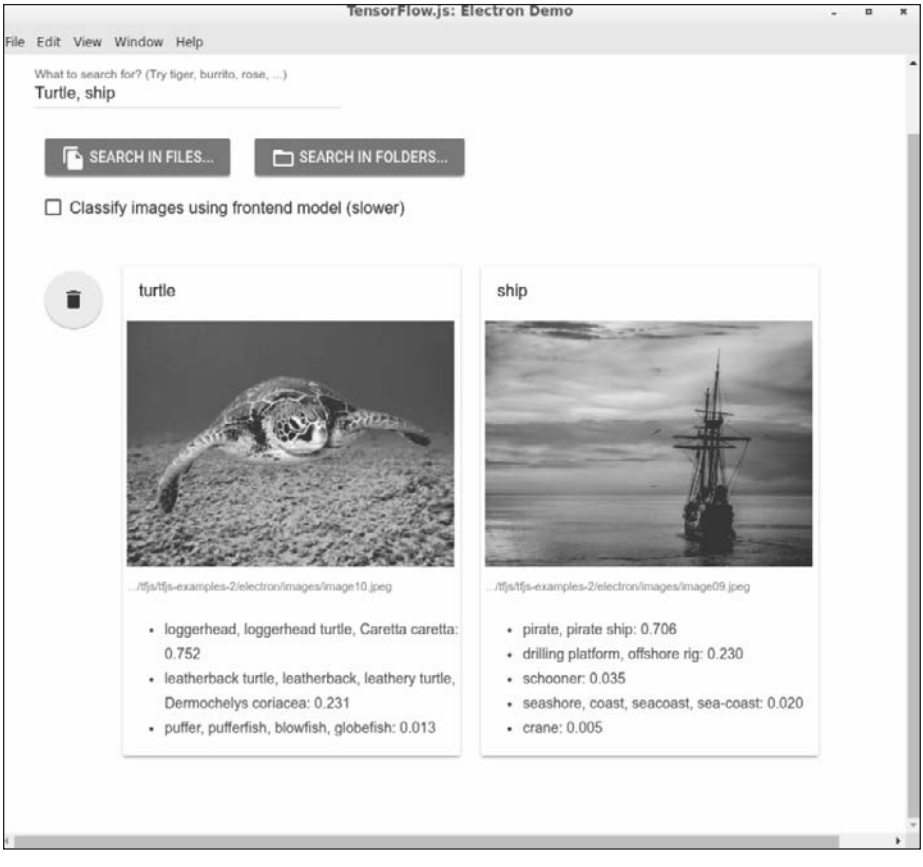


Рис. 12.7. Снимок экрана из примера приложения, использующего модель TensorFlow.js, для настольных компьютеров на основе фреймворка Electron.js (из репозитория tfjs-examples/electron)

При развертывании в клиентской части рабочая нагрузка по глубокому обучению ложится на WebGL, что вполне приемлемо в случае моделей небольшого и среднего размера либо когда задержка при выполнении вывода не играет роли. Размер пакета при этом варианте меньше, и благодаря обширной поддержке WebGL, все работает сразу же, без дополнительных действий.

Рисунок 12.8 также демонстрирует, что выбор прикладной части для вычислений в основном не зависит от загружающего и выполняющего модель JavaScript кода. Для всех трех вышеупомянутых вариантов подходит один и тот же API, что ясно показывает наш пример приложения, в котором один и тот же модуль (`ImageClassifier`

в файле `electron/image_classifier.js`) выполняет вывод в среде как прикладной, так и клиентской части. Стоит также отметить, что, хотя в примере tfjs-examples/electron показано только выполнение вывода, безусловно, TensorFlow.js можно с тем же успехом использовать и для других технологических процессов глубокого обучения, в частности создания и обучения (например, переноса обучения) модели в приложениях Electron.js.

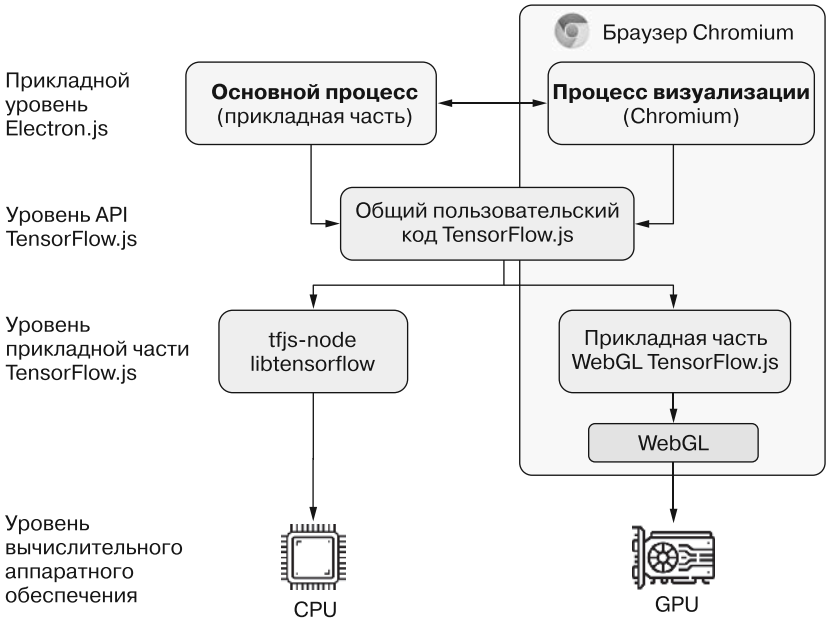


Рис. 12.8. Архитектура основанного на Electron.js приложения для настольных компьютеров, использующего TensorFlow.js для глубокого обучения с ускорением. Из основного процесса прикладной части или процесса визуализации в браузере можно вызывать различные вычислительные прикладные части TensorFlow.js. При выборе различных вариантов прикладной части модели работают на различном аппаратном обеспечении. Вне зависимости от выбранной вычислительной прикладной части, код загрузки, описания и выполнения моделей глубокого обучения в TensorFlow.js практически одинаков. Стрелки на этой схеме обозначают вызовы библиотечных функций и прочих вызываемых процедур

12.3.6. Развертывание моделей TensorFlow.js в WeChat и прочих системах плагинов мобильных приложений на основе JavaScript

Иногда основная платформа распространения мобильных приложений — не Play Store Android и не Play Store Apple, просто небольшое количество «мобильных сверхприложений», поддерживающих использование сторонних расширений внутри управляемого основным приложением пространства.

Небольшая часть этих сверхмобильных приложений создана китайскими технологическими гигантами, в частности WeChat компании Tencent, Alipay компании Alibaba и Baidu. Основной технологией, открывающей возможность создания сторонних расширений, в них служит JavaScript, так что TensorFlow.js идеально подходит для развертывания машинного обучения на этих платформах. Набор доступных в системах плагинов этих мобильных приложений API отличается от набора нативных API JavaScript, прочем так, что для развертывания на них потребуются определенные дополнительные знания и усилия.

Возьмем для примера WeChat. WeChat — наиболее распространенное приложение социальных медиа в Китае, более чем с миллиардом активных пользователей. В 2017 году WeChat запустил Mini Program — платформу, позволяющую разработчикам приложений создавать мини-программы внутри системы WeChat. Пользователи могут обмениваться этими мини-программами и устанавливать их внутри приложения WeChat во время работы. Успех этой платформы был колоссальным. Ко второму кварталу 2018 года в экосистеме WeChat было более 1 миллиона мини-программ и более 600 миллионов их активных пользователей. А также более 1,5 миллиона разработчиков приложений на этой платформе, частично вследствие популярности JavaScript.

API мини-программ WeChat нацелены на предоставление разработчикам простого доступа к датчикам мобильных устройств (камере, микрофону, акселерометру, гироскопу, GPS и т. д.). Однако нативные API предлагают лишь очень ограниченную встроенную функциональность для машинного обучения. TensorFlow.js как решение для машинного обучения для мини-программ обладает несколькими достоинствами. До него, если разработчик хотел встроить в приложение машинное обучение, он должен был работать вне среды разработки мини-программ с серверным или облачным стеком инструментов машинного обучения. Что значительно усложняло для многих разработчиков мини-программ создание кода и применение машинного обучения. Развертывание внешней инфраструктуры выходит за рамки возможностей большинства разработчиков мини-программ. А при использовании TensorFlow.js разработка кода машинного обучения происходит прямо внутри нативной среды. Более того, подобное решение на стороне клиента помогает уменьшить сетевой трафик и снизить задержку, а также использует возможности GPU-ускорения на основе WebGL.

Создавшая TensorFlow.js команда разработала мини-программу для WeChat, с помощью которой можно подключить TensorFlow.js к своей собственной мини-программе (см. https://github.com/tensorflow/tfjs-wechat). Указанный репозиторий также содержит пример мини-программы, использующей PoseNet для подписи позиций и поз людей, замеченных камерой мобильного устройства. В ней используется TensorFlow.js с ускорением на основе недавно добавленного API WebGL от WeChat. Без доступа к GPU эта модель работала бы слишком медленно для большинства приложений. А в этом плагине скорость работы модели в мини-программе WeChat будет такой же, как и у JavaScript-приложения, работающего в мобильном браузере. На самом деле мы даже заметили, что API датчиков WeChat обычно работает *быстрее*, чем его аналог в браузере.

По состоянию на конец 2019 года разработка функциональности машинного обучения для плагинов сверхмобильных приложений — совершенно нехоженая территория. Высокое быстродействие порой требует определенной помощи со стороны самой платформы. Тем не менее это оптимальный способ для предоставления доступа к своему приложению сотням миллионов человек, для которых сверхмобильное приложение *и есть* Интернет.

12.3.7. Развертывание моделей TensorFlow.js на одноплатных компьютерах

Для многих веб-разработчиков сама идея развертывания программы на автономных одноплатных компьютерах представляется чуждой и технически слишком сложной. Однако благодаря успеху Raspberry Pi проектирование и создание простых аппаратных устройств невероятно упростилось. Одноплатные компьютеры предоставляют платформу для создания «умных» устройств без необходимости сетевого соединения с облачными серверами или громоздких, дорогостоящих компьютеров. Одноплатные компьютеры можно использовать для приложений системы безопасности, контроля интернет-трафика, управления ирригацией почв — пределов их возможностям нет.

Многие из таких одноплатных компьютеров предоставляют контакты универсального интерфейса ввода/вывода (general-purpose input-output, GPIO) для простого подключения аппаратных систем управления и включают полный дистрибутив Linux, с помощью которых преподаватели, разработчики и хакеры могут создавать широкий спектр интерактивных устройств. Одним из популярных языков программирования для создания подобных устройств быстро стал JavaScript. Для взаимодействия на нижнем, физическом уровне разработчики могут использовать такие библиотеки node, как rpi-gpio, все — на JavaScript.

В TensorFlow.js, для поддержки таких пользователей есть в настоящее время две среды выполнения для этих встроенных устройств ARM: tfjs-node (CPU[1]) и tfjs-headless-nodegl (GPU). С помощью этих двух прикладных частей можно использовать на этих устройствах всю библиотеку TensorFlow.js. Разработчики могут выполнять вывод с помощью готовых моделей или обучать свои собственные модели, все — на аппаратном обеспечении такого устройства!

В недавно появившихся устройствах, таких как NVIDIA Jetson Nano и Raspberry Pi 4, появилась технология *«система на микросхеме»* (system-on-chip, SoC) с современным графическим стеком. Применяемый в ядре TensorFlow.js код WebGL позволяет использовать GPU этих устройств. С помощью автономного пакета WebGL (tfjs-headless-nodegl) пользователи могут запускать TensorFlow.js на Node.js с ускорением исключительно за счет GPU этих устройств (рис. 12.9). Благодаря

[1] Для использования возможностей CPU с ускорением ARM NEON следует применять на этих устройствах пакет tfjs-node, обеспечивающий поддержку как архитектуры ARM32, так и ARM64.

делегированию GPU выполнения TensorFlow.js разработчики могут использовать CPU для управления остальными частями устройств.

Безопасность модели и данных в случае развертывания на одноплатном компьютере очень высока. Вычисления производятся непосредственно на устройстве, а значит, данные не нужно передавать на сторонние, неподконтрольные устройства. Для защиты модели на случай несанкционированного доступа к физическому устройству можно использовать шифрование.

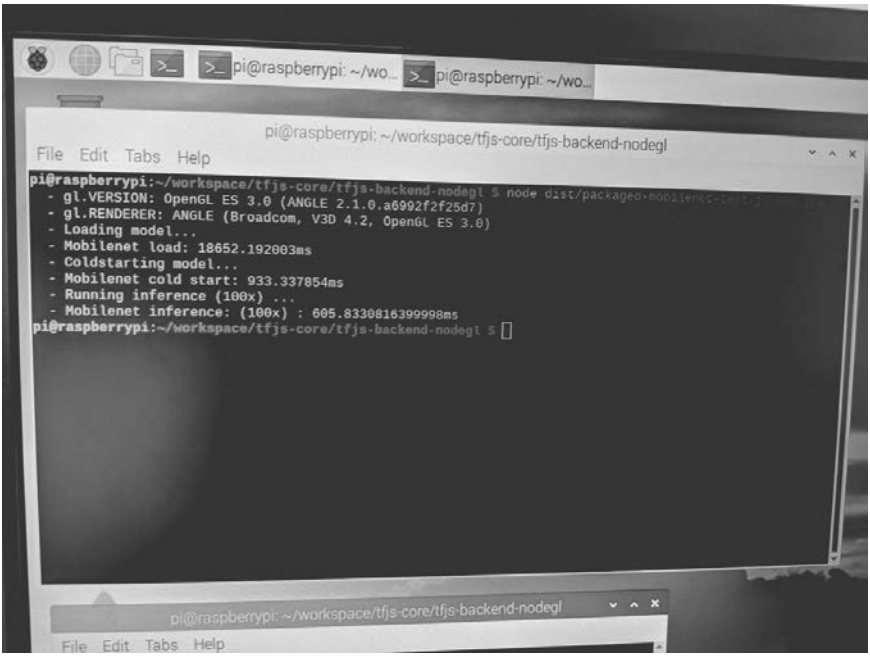


Рис. 12.9. Работа модели MobileNet в TensorFlow.js с помощью автономного WebGL на Raspberry Pi 4

Развертывание на одноплатных компьютерах — пока еще очень новая область применения JavaScript вообще и TensorFlow.js в частности, открывающая, однако, путь для широкого спектра приложений, для которых прочие варианты развертывания не подходят.

12.3.8. Краткая сводка вариантов развертывания

В этом разделе мы охватили несколько различных способов предоставления аудитории пользователей доступа к системе машинного обучения на основе TensorFlow.js (их краткую сводку вы можете найти в табл. 12.4). Надеемся, что мы зажгли искру вашего воображения и заставили вас мечтать о самых дерзновенных способах применения

этой технологии! Экосистема JavaScript чрезвычайно обширна, и в будущем системы на основе машинного обучения будут работать в областях, о которых сегодня мы даже не мечтаем.

Таблица 12.4. Возможные среды развертывания моделей TensorFlow.js и средства аппаратного ускорения в каждой из них

Среда развертывания	Поддержка аппаратного ускорения
Браузер	WebGL
Сервер Node.js	CPU с поддержкой многопоточности и SIMD; GPU с поддержкой CUDA
Браузерный плагин	WebGL
Межплатформенное приложение для настольных компьютеров (например, на основе Electron)	WebGL, CPU с поддержкой многопоточности и SIMD или GPU с поддержкой CUDA
Межплатформенное мобильное приложение (например, на основе React Native)	WebGL
Плагин для мобильного приложения (такого как WeChat)	Мобильный WebGL
Одноплатный компьютер (например, Raspberry Pi)	GPU или ARM NEON

Материалы для дальнейшего изучения

- *Baylor D. et al.* TFX: A TensorFlow-Based Production-Scale Machine Learning Platform // KDD 2017: www.kdd.org/kdd2017/papers/view/tfx-a-tensorflow-based-production-scale-machine-learning-platform.
- *Krishnamoorthi R.* Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper // June 2018: https://arxiv.org/pdf/1806.08342.pdf.
- *Larsen R. M., Shpeisman T.* TensorFlow Graph Optimization: https://ai.google/research/pubs/pub48051.

Упражнения

1. В главе 10 мы обучили вспомогательный классификатор GAN (ACGAN) на наборе данных MNIST для генерации поддельных изображений цифр нужного класса. А именно, речь идет о примере из каталога mnist-acgan репозитория tfjs-examples. Общий размер генератора обученной модели составлял примерно 10 Мбайт, большую часть которых занимали весовые коэффициенты, хранимые в виде 32-битных чисел с плавающей точкой. Было бы заманчиво произвести кванто-

вание весов этой модели после обучения — для ускорения загрузки страницы. Однако перед этим необходимо убедиться, что подобное квантование не приведет к существенному ухудшению качества генерируемых изображений. Проверьте, насколько приемлем в этом смысле каждый из вариантов 16- и 8-битного квантования. Воспользуйтесь технологическим процессом `tensorflowjs_converter`, описанным в подразделе 12.2.1. Какими критериями оценки качества сгенерированных изображений MNIST вы воспользуетесь в данном случае?

2. Одно из преимуществ моделей TensorFlow, работающих в виде расширения Chrome, — возможность управлять самим браузером Chrome. В примере speech-commands из главы 4 мы показали, как распознавать голосовые команды с помощью сверточной модели. API расширений Chrome предоставляет возможность выполнения запросов и перехода с вкладки на вкладку. Попробуйте встроить модель speech-commands в расширение и научите его распознавать фразы next tab (следующая вкладка) и previous tab (предыдущая вкладка). Воспользуйтесь результатами работы этого классификатора для управления тем, какая вкладка браузера активна в настоящий момент.
3. Инфобокс 12.3 описывает, как правильно измерять время выполнения вызова метода `predict()` модели TensorFlow.js (выполнения вывода), и нюансы этого процесса. В данном упражнении загрузите модель MobileNetV2 в TensorFlow.js (чтобы освежить свои знания об этом, см. пример simple-object-detection в разделе 5.2) и измерьте длительность выполнения его вызова метода `predict()`.
   А. Сначала сгенерируйте тензор изображений со случайными значениями формы `[1, 224, 224, 3]` и измерьте длительность вывода модели на его основе, следуя изложенному в инфобоксе 12.3 алгоритму. Сравните результаты такого хронометража с вызовом `array()` или `data()` для выходного тензора (или без него). Какой из вариантов выполняется быстрее? В каком случае время измеряется правильно?
   Б. Постройте график значений длительности выполнения при проведении правильного измерения 50 раз в цикле с помощью линейной диаграммы из библиотеки tfjs-vis (глава 7) и почувствуйте колебания этих значений. Заметно ли, что первые несколько измерений существенно отличаются от остальных? С учетом этого наблюдения обсудите важность «прогрева» при оценке быстродействия.
   В. В отличие от задач А и Б возьмите вместо сгенерированного случайным образом входного тензора настоящий тензор изображения (например, полученный из элемента `img` с помощью метода `tf.browser.fromPixels()`) и повторите измерения из задачи Б. Влияет ли существенно содержимое входного тензора на хронометраж?
   Г. Вместо выполнения вывода для отдельного примера данных (размер батча = 1) попробуйте увеличить размер батча до 2, 3, 4 и т. д., вплоть до относительно большого числа, скажем 32. Является ли зависимость между средним временем выполнения вывода и размером батча монотонно возрастающей? Линейной?

Резюме

- Разумный подход к тестированию кода машинного обучения столь же важен, как и к тестированию не относящегося к нему кода. Однако старайтесь не сосредотачивать свое внимание исключительно на частных случаях или на контроле «золотых значений» для предсказаний. Вместо этого старайтесь тестировать базовые свойства модели, например характеристики входных и выходных данных. Более того, помните, что весь код предварительной обработки данных, предваряющий систему машинного обучения, — такой же «обычный» код и его следует тестировать соответствующим образом.
- Оптимизация скорости скачивания и выполнения вывода — важный фактор успеха развертывания моделей TensorFlow.js на стороне клиента. Благодаря возможности квантования весовых коэффициентов после обучения команды `tensorflowjs_converter` можно сократить общий размер модели, причем в некоторых случаях без какой-либо существенной потери степени безошибочности вывода. Возможность преобразования графов вычислений `tensorflowjs_converter` позволяет ускорить выполнение вывода на основе модели посредством преобразований графов, в частности слияния операций. Мы настоятельно рекомендуем вам тестировать и использовать во всей полноте обе эти методики оптимизации моделей при развертывании моделей TensorFlow.js для промышленной эксплуатации.
- Обученная и оптимизированная модель — далеко не все, что требуется для приложения машинного обучения. Необходимо еще найти способ интегрировать ее в сам программный продукт. Чаще всего приложения TensorFlow.js развертываются внутри веб-страниц, но это лишь один из множества разнообразных сценариев развертывания, каждый со своими достоинствами и недостатками. Модели TensorFlow.js могут работать в виде браузерных расширений, в нативных мобильных приложениях, в виде нативных приложений для настольных компьютеров и даже на одноплатном аппаратном обеспечении наподобие Raspberry Pi.

13 Резюме, заключительные слова и дальнейшие источники информации

В этой главе

- Обзор понятий и идей, связанных с ИИ и глубоким обучением.
- Краткий обзор обсуждавшихся в этой книге типов алгоритмов глубокого обучения, сферы их применения и реализация их в TensorFlow.js.
- Предобученные модели из экосистемы TensorFlow.js.
- Текущие ограничения глубокого обучения; прогноз тенденций глубокого обучения на ближайшие годы.
- Руководство по дальнейшему расширению ваших знаний глубокого обучения и тому, как не отстать от прогресса в этой быстро меняющейся сфере.

Это последняя глава нашей книги. В предыдущих главах мы совершили путешествие по стране современного глубокого обучения благодаря предоставленному нам TensorFlow.js транспорту и вашей собственной усердной работе с книгой. Надеемся, в этом путешествии вы освоили много новых понятий и обрели немало новых навыков. Время оглянуться и посмотреть с высоты птичьего полета на всю эту страну в целом, равно как и освежить в памяти некоторые наиболее важные идеи. В этой, последней главе мы подведем итоги и напомним основные понятия, параллельно открывая вам новые горизонты, выходящие далеко за пределы относительно простых идей, изученных вами ранее. Мы хотим убедиться, что вы готовы к последующему самостоятельному путешествию.

Мы начнем с общей картины всего, что вы должны вынести из этой книги, и освежим таким образом в вашей памяти некоторые из изученных концепций. Далее мы приведем обзор важнейших ограничений глубокого обучения. Для должного использования любого инструмента необходимо знать не только на что он *способен*, но и на что он *не способен*. Завершается эта глава перечнем ресурсов и стратегий, которые позволят вам не только углубить знания и навыки глубокого обучения и ИИ в экосистеме JavaScript, но и оставаться на уровне последних достижений в этой сфере.

13.1. Обзор ключевых понятий

В этом разделе кратко подытожен «сухой остаток» этой книги. Начнем с общего ландшафта ИИ, а в завершение расскажем, почему сочетание глубокого обучения и JavaScript открывает уникальные захватывающие возможности.

13.1.1. Различные подходы к ИИ

Прежде всего, глубокое обучение — вовсе не синоним ИИ или даже машинного обучения. *Искусственный интеллект* (artificial intelligence) — обширная сфера исследований с долгой историей. В целом искусственный интеллект можно описать как «все попытки автоматизации процесса познания» — другими словами, как автоматизацию процесса мышления. Он простирается от таких простейших вещей, как электронные таблицы Excel, до самых передовых проектов, таких как человекоподобные роботы, способные ходить и говорить.

Машинное обучение (machine learning) — одна из многих подобластей искусственного интеллекта, нацеленная на автоматическую разработку программ (так называемых *моделей*) путем воздействия на них обучающих данных. Процесс превращения данных в программу (модель) называется *обучением* (learning). И хотя машинное обучение уже существует довольно давно (по крайней мере несколько десятилетий), широко применять на практике его начали только в 1990-х.

Глубокое обучение (deep learning) — одна из многих форм машинного обучения. В глубоком обучении модели состоят из множества последовательно применяемых шагов преобразований представлений (отсюда название «глубокое»). Эти операции организуются в модули, называемые *слоями* (layers). Модели глубокого обучения обычно состоят из последовательности (в общем случае — графа) множества слоев. Эти слои параметризуются *весовыми коэффициентами* (*весами*, weights) — числовыми значениями, обновляемыми в ходе обучения, на основе которых входной сигнал слоя преобразуется в выходной. В этих весах заключены усвоенные моделью во время обучения «знания». Целью процесса обучения главным образом является поиск хорошего набора значений весовых коэффициентов.

И хотя глубокое обучение — лишь один из множества подходов машинного обучения, по сравнению с прочими подходами его успех был ошеломляющим. Перечислим основные причины такого успеха глубокого обучения.

13.1.2. Почему глубокое обучение выделяется из всех прочих подобластей машинного обучения

За какие-то несколько лет глубокое обучение позволило достичь выдающихся успехов в решении множества задач, считавшихся ранее исключительно трудными для компьютеров, особенно в сфере машинного восприятия — а именно, выделения полезной информации из изображений, аудио-, видеоданных и тому подобных типов сенсорных входных данных с весьма высокой степенью безошибочности. При достаточном количестве обучающих данных (в частности, *маркированных* обучающих данных) стало возможно выделять из сенсорных входных данных практически всю информацию, которую только способен извлечь живой человек, иногда даже с меньшим количеством ошибок, чем люди. Поэтому иногда говорят, что глубокое обучение более или менее «решило задачу машинного восприятия», хотя это справедливо лишь при довольно узком понимании термина «восприятие» (см. перечень ограничений глубокого обучения в подразделе 13.2.5).

Глубокое обучение вследствие беспрецедентного успеха само по себе привело к наступлению третьего и самого масштабного *лета искусственного интеллекта*, называемого также революцией глубокого обучения, — периода повышенного интереса, инвестиций и шумихи в сфере ИИ. Закончится ли этот период в ближайшем будущем и что будет дальше — можно только гадать. Но ясно одно: в отличие от предыдущих периодов расцвета искусственного интеллекта глубокое обучение принесло огромную пользу множеству высокотехнологических компаний, сделав возможными — на уровне человека или лучше — классификацию изображений, обнаружение объектов, распознавание речи, интеллектуальных виртуальных помощников, обработку текстов на естественном языке, машинный перевод, рекомендательные системы, беспилотные автомобили и многое, многое другое. Шумиха постепенно уляжется (что вполне закономерно), но долговременное влияние на технологии и экономические выгоды останутся. В этом смысле глубокое обучение аналогично Интернету: чрезмерная шумиха в течение нескольких лет, а потому необоснованные ожидания и чрезмерные инвестиции, но в долгосрочной перспективе Интернет — настоящая революция, влияющая на многие сферы технологии и меняющая весь наш жизненный уклад.

Особый оптимизм в отношении глубокого обучения вызывает тот факт, что, даже если за ближайшее десятилетие никакого особого прогресса в нем не наметится, само применение уже существующих методик ко всем возможным задачам поменяет весь расклад во многих отраслях промышленности (интернет-рекламе, финансах, промышленной автоматизации и технологиях специальных возможностей для людей с физическими ограничениями — и это только небольшая часть). Глубокое обучение — настоящая революция и развивается в настоящее время с невероятной быстротой благодаря растущим в геометрической прогрессии вложениям в ресурсы и персонал. С нашей точки зрения, будущее представляется весьма радужным, хотя краткосрочные перспективы, возможно, не столь оптимистичны, как кажется: раскрытие полного потенциала глубокого обучения потребует никак не меньше десятка лет.

13.1.3. Общая картина глубокого обучения

В глубоком обучении больше всего удивляет его простота, если учесть, насколько хорошо оно работает, в то время как предшествовавшие ему гораздо более сложные методики машинного обучения демонстрировали намного худшие результаты. Десять лет назад никто не ожидал таких потрясающих результатов в задачах машинного восприятия от простых параметрических моделей, обученных с помощью градиентного спуска. Теперь же оказалось, что для успеха хватит всего лишь достаточно большой параметрической модели, обученной на основе градиентного спуска, и большого набора обучающих примеров данных. Как однажды сказал о Вселенной Ричард Фейнман, «она не сложна, просто очень велика»[1].

В глубоком обучении все данные представляются в виде рядов чисел — другими словами, *векторов*. Вектор можно считать *точкой* в *геометрическом пространстве*. Входные сигналы моделей (табличные данные, изображения, текст и т. д.) сначала преобразуются в векторы, то есть набор точек во входном векторном пространстве. Аналогичным образом целевые величины (метки) также преобразуются в соответствующие векторы — набор точек в целевом векторном пространстве. Цепочка слоев нейронной сети осуществляет сложное геометрическое преобразование, состоящее из ряда простых геометрических преобразований, которое ставит точкам во входном векторном пространстве точки в целевом векторном пространстве. Параметрами этого преобразования служат весовые коэффициенты слоев, обновляемые на каждом шаге в зависимости от того, насколько хорошие результаты демонстрирует преобразование в настоящий момент. Ключевая характеристика этого геометрического преобразования, благодаря которой и возможен градиентный спуск, — его *дифференцируемость*.

13.1.4. Ключевые технологии, благодаря которым возможно глубокое обучение

Нынешняя революция глубокого обучения не началась за один день, а, как и все прочие революции, является результатом постепенного накопления множества факторов — сначала медленного, а потом резко ускорившегося, когда накопилась критическая их масса. В случае глубокого обучения можно отметить следующие ключевые факторы.

- Постепенные разработки новых алгоритмов, сначала нечастые, растянувшиеся на два десятилетия[2], а затем существенно ускорившиеся после 2012 года, когда в этом направлении начали вести более активную исследовательскую работу[3].

---

[1] Ричард Фейнман, интервью «Мир с другой точки зрения» (The World from Another Point of View // Yorkshire Television, 1972.)

[2] Начиная с изобретения обратного распространения ошибки Румельхартом, Хинтоном и Уильмсом, сверточных слоев Ле Куном и Бенжио, а также сверточных сетей Грейвсом и Шмидтхубером.

[3] Например, появились усовершенствованные методы инициализации весовых коэффициентов, новые функции активации, дропаут, нормализация по батчам, остаточные связи.

- Доступность большого количества маркированных данных для широкого диапазона типов входных данных, включая сенсорные (изображения, аудио и видео), числовые и текстовые, что позволило обучать большие модели на достаточных объемах данных. А все благодаря широкому распространению мобильных устройств, в результате чего начался настоящий бум Интернета для конечных пользователей, а также закону Мура применительно к запоминающим носителям.
- Доступность дешевого и быстродействующего распараллеливаемого аппаратного обеспечения, особенно GPU от компании NVIDIA, — сначала перепрофилированных для параллельных вычислений игровых GPU, а затем и микросхем, специально предназначенных для глубокого обучения.
- Полный стек программного обеспечения с открытым исходным кодом, благодаря которому все эти вычислительные возможности стали доступны для множества разработчиков и студентов (причем не нужно было изучать весь колоссальный объем сложностей, лежащих в их основе): язык CUDA, языки программирования шейдеров WebGL браузеров, а также такие фреймворки, как TensorFlow.js, TensorFlow и Keras, автоматически осуществлявшие дифференцирование и предоставлявшие высокоуровневые, удобные в использовании «строительные блоки» — слои, функции потерь и оптимизаторы. Глубокое обучение превращается из монополии узких специалистов (исследователей, аспирантов в сфере ИИ и инженеров с хорошим уровнем теоретической подготовки) в инструмент, доступный любому программисту. В этом смысле TensorFlow.js — образцовый фреймворк, в котором сходятся вместе две обладающие огромным потенциалом процветающие экосистемы: межплатформенная экосистема JavaScript и динамично развивающаяся экосистема глубокого обучения.

Яркое проявление обширного влияния революции глубокого обучения — его слияние с технологическими стеками, отличными от родного (экосистемы C++/Python и сферы численных расчетов). Основной пример этого: взаимное влияние его и экосистемы JavaScript — главная тема данной книги. В следующем разделе мы еще раз перечислим основные причины того, почему глубокое обучение в мире JavaScript открывает новые возможности и перспективы.

13.1.5. Сферы применения и возможности, открываемые благодаря глубокому обучению на JavaScript

Основная цель обучения модели ГО — дальнейшее применение ее конечными пользователями. Многие типы входных данных, например изображения с веб-камеры, звуки с микрофона, текстовые и жестовые входные данные от пользователя, генерируются непосредственно на клиенте и доступны там. JavaScript, вероятно, наиболее развитый и повсеместно доступный язык программирования и экосистема для создания программ, работающих на стороне клиента. Один и тот же код на JavaScript можно развернуть в виде веб-страниц и UI на множестве устройств

и платформ. API WebGL браузера позволяет производить параллельные вычисления на самых разнообразных GPU на различных платформах, что широко используется в TensorFlow.js. Благодаря этому JavaScript — прекрасный вариант для развертывания моделей глубокого обучения. TensorFlow.js предоставляет утилиту для преобразования моделей, обученных в таких популярных фреймворках Python, как TensorFlow и Keras, в подходящий для веб-развертывания формат и развертывания их на веб-страницах для выполнения вывода и переноса обучения.

Помимо удобства развертывания, есть и несколько других преимуществ тонкой настройки моделей глубокого обучения и выполнения вывода на их основе с помощью JavaScript.

- По сравнению с выполнением вывода на стороне сервера, при выполнении вывода на стороне клиента отсутствует задержка из-за передачи данных туда и обратно, что повышает доступность и плавность работы с точки зрения пользователей.
- Благодаря выполнению вычислений на клиенте с локальным ускорением на основе GPU для глубокого обучения не требуется управлять ресурсами GPU на сервере, что значительно снижает сложность соответствующего стека технологий и затраты на его сопровождение.
- Улучшается защита персональной информации пользователей благодаря тому, что данные и результаты выполнения вывода не покидают клиентской машины. Что играет важную роль, в частности, в сфере здравоохранения и моды.
- Браузер и прочие среды UI на основе JavaScript наглядны и интерактивны, обеспечивая уникальные возможности для визуализации и изучения нейронных сетей.
- TensorFlow.js поддерживает не только выполнение вывода, но и обучение. Что открывает возможности переноса обучения и тонкой настройки на стороне клиента, а значит, и улучшения подгонки моделей машинного обучения под конкретного пользователя.
- В браузере JavaScript предоставляет платформонезависимый API для доступа к сенсорам устройств, например к веб-камерам и микрофонам, в результате чего разработка использующих данные с этих сенсоров межплатформенных приложений значительно ускоряется.

Помимо господствующей позиции среди языков на стороне клиента, JavaScript может немало предложить и на стороне сервера. В частности, один из очень популярных фреймворков создания серверных приложений на JavaScript — Node.js. С помощью основанной на Node.js версии TensorFlow.js (tfjs-node) можно обучать модели глубокого обучения и выполнять на их основе вывод за пределами браузера, а значит, без соответствующих ограничений ресурсов. Благодаря доступу к обширной экосистеме Node.js стек технологий для разработчиков сильно упрощается. Причем все это можно реализовать фактически с помощью того же самого кода TensorFlow.js, что и на стороне клиента, приближая нас к парадигме «написать один раз, выполнять где угодно», как было показано в нескольких примерах из книги.

13.2. Краткий обзор технологического процесса глубокого обучения и алгоритмов в TensorFlow.js

С обзором истории покончено, и мы можем заняться техническими аспектами TensorFlow.js. В этом разделе мы опишем общий технологический процесс решения задач машинного обучения, указав несколько важнейших нюансов и распространенных подводных камней. А затем перечислим различные «кирпичики» (слои) нейронных сетей, обсуждавшиеся в этой книге. Кроме того, мы рассмотрим предобученные модели из экосистемы TensorFlow.js, с помощью которых можно ускорить процесс разработки. В завершение раздела мы приведем спектр задач машинного обучения, которые можно решить с помощью указанных «кирпичиков». Подумайте, как написанные на TensorFlow.js глубокие нейронные сети могут помочь в решении стоящих перед вами задач машинного обучения.

13.2.1. Универсальный технологический процесс машинного обучения с учителем

Возможности глубокого обучения огромны. Но, как ни странно, зачастую больше всего времени и усилий из процесса машинного обучения занимают этапы, предваряющие собственно проектирование и обучение моделей (а в случае предназначенных для промышленной эксплуатации моделей также и следующие за ними). На этих непростых этапах необходимо разобраться в предметной области задачи достаточно хорошо для того, чтобы определить, какие данные нужны, какие предсказания можно выполнить с достаточной степенью безошибочности и обобщения, какое место модель машинного обучения займет в общем программном решении, предназначенном для конкретной задачи, а также как оценить качество работы модели. И хотя без этого невозможен успех на практике никакой модели машинного обучения, автоматизировать эти действия библиотека программных модулей вроде TensorFlow.js не способна. Напомним вам вкратце, как выглядит типовой технологический процесс машинного обучения.

1. *Выяснить, является ли машинное обучение правильным подходом.* Во-первых, необходимо обдумать, хорошо ли подходит машинное обучение для решения поставленной задачи, и переходить к следующим шагам только в том случае, если ответ на этот вопрос положительный. В некоторых случаях не основанный на машинном обучении подход даст ничуть не худшие, а то и лучшие результаты при меньших затратах.
2. *Сформулировать задачу машинного обучения.* Определите, какие у вас имеются данные и что необходимо предсказать на их основе.
3. *Убедиться, достаточно ли имеющихся данных.* Выясните, достаточно ли имеющегося объема данных для обучения модели. Возможно, придется собрать

дополнительные данные или нанять людей для маркирования вручную немаркированного набора данных.

4. *Найти надежную меру успешности работы обученной модели относительно цели*. В простых задачах такой мерой может служить степень безошибочность предсказаний, но во многих случаях потребуются более сложные предметно-ориентированные метрики.
5. *Подготовка процесса оценки*. Необходимо спроектировать процесс оценки качества работы модели. В частности, необходимо разбить данные на три однородных, но непересекающихся множества: обучающий, проверочный и контрольный наборы данных. Важно, чтобы метки проверочного и контрольного наборов данных не попали в обучающие данные. Например, в случае временны*х* прогнозов проверочные и контрольные данные должны браться из промежутков времени, следующих за обучающими данными. Код предварительной обработки данных должен полностью охватываться тестами для защиты от подобных ошибок (см. раздел 12.1).
6. *Векторизация данных*. Данные необходимо преобразовать в тензоры (*n*-мерные массивы) — лингва франка моделей машинного обучения в таких фреймворках, как TensorFlow.js и TensorFlow. Чтобы приспособить тензоризованные данные для конкретных моделей, нередко приходится подвергать их предварительной обработке, например нормализации.
7. *Разработать модель, работающую лучше, чем эталонный подход, основанный просто на здравом смысле*. Разработать модель, работающую лучше, чем эталонная версия, не основанная на машинном обучении (наподобие предсказания средних по населению показателей для задачи регрессии или наподобие задачи предсказания последней точки данных в ряде данных), продемонстрировав таким образом, что машинное обучение действительно может принести вам пользу. Что не всегда так (см. пункт 1).
8. *Разработать модель с достаточными разрешающими возможностями, причем чрезмерно подогнанную к обучающим данным*. Необходимо постепенно масштабировать вверх архитектуру своей модели, подбирая гиперпараметры и вводя регуляризацию. Учтите: вносить изменения на основе степени безошибочности следует только в проверочный набор данных, необучающий и не контрольный. Помните, ваша цель — получить в итоге модель, чрезмерно подогнанную к обучающему набору данных (степень безошибочности на обучающем наборе данных чуть лучше, чем на проверочном), а значит, найти грань между недостаточными и чрезмерными разрешающими возможностями модели. Лишь после этого следует использовать регуляризацию и прочие методики сокращения переобучения.
9. *Подбор гиперпараметров*. При подборе гиперпараметров необходимо отслеживать переобучение на проверочном наборе данных. Поскольку гиперпараметры подбираются исходя из работы модели на проверочном наборе, их значения будут слишком хорошо подходить именно для него, а потому плохо обобщаться на прочие данные. Получение несмещенной оценки безошибочности модели после подбора гиперпараметров — задача контрольного набора данных. Поэтому при подборе гиперпараметров использовать контрольный набор данных нельзя.

10. *Проверка и оценка обученной модели.* Как мы обсуждали в разделе 12.1, необходимо проверить модель на как можно более свежем оценочном наборе данных и определить, отвечает ли степень безошибочности предсказания заранее выбранному критерию для промышленной эксплуатации. Кроме того, необходимо произвести углубленный анализ качества работы модели на различных срезах (подмножествах) данных, чтобы выявить нежелательные систематические ошибки и убедиться в отсутствии дискриминации (например, сильно отличающейся степени безошибочности на различных срезах данных)[1]. И лишь если модель удовлетворяет этим критериям оценки, следует переходить к последнему шагу.
11. *Оптимизация и развертывание модели.* Выполните оптимизацию модели для сокращения ее размера и ускорения вывода. После чего можно произвести развертывание модели в среде, предназначенной для выдачи результатов пользователю, например на веб-странице, в мобильном приложении или конечной точке HTTP-сервиса (см. раздел 12.3).

Это алгоритм обучения с учителем, применяемый для решения множества реальных задач. В этой книге также описывались технологические процессы и других типов машинного обучения: переноса обучения (с учителем), обучения с подкреплением (RL) и генеративного глубокого обучения. Технологические процессы переноса обучения с учителем (глава 5) и обычного обучения с учителем совпадают, разве что в основе архитектуры модели и шагов обучения лежит предобученная модель, поэтому обучающих данных обычно нужно меньше, чем при обучении с нуля. Цель генеративного глубокого обучения отличается от обучения с учителем, а именно: необходимо создать фиктивные примеры данных, как можно больше напоминающие настоящие. На практике существуют методики, позволяющие свести обучение генеративных моделей к обучению с учителем, как мы видели в примерах VAE и GAN в главе 9. Постановка задачи RL, с другой стороны, совершенно иная, так что и технологический процесс резко отличается — основную роль в нем играют среда, агент, действия, структура вознаграждений и алгоритм или типы используемых для решения задачи моделей. Основные понятия и алгоритмы RL кратко описаны в главе 11.

13.2.2. Типы моделей и слоев в TensorFlow.js: краткий справочник

Все многочисленные нейронные сети, описываемые в этой книге, можно разбить на три семейства: плотносвязные сети (называемые также многослойными перцептронами (MLP)), сверточные сети и рекуррентные сети. Каждый практический специалист по глубокому обучению должен быть знаком с этими основными семействами сетей. Каждый из этих типов сетей подходит для своих входных данных: архитектура сети (MLP, сверточной или рекуррентной) кодирует определенные

[1] Дискриминация (fairness) в машинном обучении — лишь зарождающаяся сфера исследований; см. более подробное обсуждение по следующей ссылке: http://mng.bz/eD4Q.

допущения относительно структуры входных данных, образующие пространство гипотез, в котором посредством обратного распространения ошибки и подбора гиперпараметров происходит поиск хорошей модели. Подойдет ли конкретная архитектура для имеющейся задачи — зависит всецело от того, насколько структура данных соответствует допущениям, заложенным в архитектуру сети.

Все эти разные типы сетей можно комбинировать, подобно кубикам LEGO, создавая более сложные сети, подходящие для различных типов данных. В некотором смысле слои глубокого обучения — кубики LEGO для обработки различных видов информации. Вот краткий список основных типов входных данных и подходящей для них архитектуры сетей.

- Векторные данные (без временно́й или последовательной упорядоченности) — MLP (плотные слои).
- Данные изображений (черно-белых, в оттенках серого или цветных) — двумерные сверточные сети.
- Аудиоданные в виде спектрограмм — двумерные сверточные сети или RNN.
- Текстовые данные — одномерные сверточные сети или RNN.
- Данные временных рядов — одномерные сверточные сети или RNN.
- Объемные пространственные данные (например, определенные виды медицинских снимков) — трехмерные сверточные сети.
- Видеоданные (последовательности изображений) — либо трехмерные сверточные сети (при необходимости захвата движений), либо сочетание двумерной сверточной сети для выделения признаков с последующей RNN или одномерной сверточной сетью для обработки полученной при этом последовательности признаков.

Давайте теперь рассмотрим каждое из этих трех основных семейств архитектур сетей подробнее и обсудим, для каких задач они подходят и как использовать их с помощью TensorFlow.js.

Плотносвязные сети и многослойные перцептроны

Понятия *«плотносвязная сеть»* (densely connected network) и «многослойный перцептрон» (multilayer perceptron, MLP), по сути, синонимы, разве что плотносвязная сеть может включать всего один слой, а многослойный перцептрон должен состоять как минимум из скрытого и выходного слоев. Ради краткости мы будем называть *MLP* все модели, состоящие преимущественно из плотных слоев. Подобные сети специализируются на обработке неупорядоченных векторных данных (например, числовых признаков в задачах обнаружения фишинговых веб-сайтов и предсказания цен на недвижимость). Каждый из плотных слоев пытается смоделировать взаимосвязь между всеми возможными парами входных признаков и своими выходными функциями активации, что достигается посредством матричного умножения ядра плотного слоя на входной вектор (с последующим прибавлением вектора смещения и функции активации). А поскольку все входные признаки влияют на все выход-

ные функции активации, то подобные слои и основанные на них сети называются *плотносвязными* (densely connected) или, как называют их некоторые авторы, *полносвязными* (fully connected). Тем самым они отличаются от прочих типов архитектуры (сверточных сетей и RNN), в которых выходной элемент может зависеть только от некоторого подмножества элементов входных данных.

Чаще всего MLP используют для категориальных данных (скажем, в которых входные признаки представляют собой список атрибутов, как в задаче обнаружения фишинговых сайтов). Кроме того, они часто используются в качестве завершающих выходных слоев большинства нейронных сетей, предназначенных для классификации и регрессии, в которых сверточные или рекуррентные слои выполняют выделение признаков, которые подают затем на вход подобных MLP. Например, обсуждавшиеся в главах 4 и 5 двумерные сверточные сети все завершают одним или двумя плотными слоями, как и рекуррентные сети из главы 9.

Давайте вкратце вспомним, как выбирается функция активации выходного слоя MLP для различных видов задач обучения с учителем. Для бинарной классификации итоговый выходной слой MLP должен включать ровно один нейрон и использовать сигма-функцию активации. В качестве функции потерь при обучении подобного бинарного классификатора на основе MLP должна использоваться `binaryCrossentropy`. Примеры данных в обучающем наборе должны содержать бинарные метки (метки со значением 0 или 1). Соответствующий код TensorFlow.js выглядит вот так:

```
import * as tf from '@tensorflow/tfjs';

const model = tf.sequential();
model.add(tf.layers.dense({units: 32, activation: 'relu', inputShape:
    [numInputFeatures]}));
model.add(tf.layers.dense({units: 32, activation: 'relu'}));
model.add(tf.layers.dense({units: 1: activation: 'sigmoid'}));
model.compile({loss: 'binaryCrossentropy', optimizer: 'adam'});
```

Для выполнения однозначной многоклассовой классификации (в которой каждому примеру данных соответствует ровно один из множества возможных классов) последовательность слоев должна завершаться плотным слоем с многомерной логистической функцией активации и количеством нейронов, равным числу классов. При унитарном кодировании целевых признаков в качестве функции потерь следует использовать `categoricalCrossentropy`, если же они представляют собой целочисленные индексы, то `sparseCategoricalCrossentropy`. Например:

```
const model = tf.sequential();
model.add(tf.layers.dense({units: 32, activation: 'relu', inputShape:
    [numInputFeatures]});
model.add(tf.layers.dense({units: 32, activation: 'relu'});
model.add(tf.layers.dense({units: numClasses: activation: 'softmax'});
model.compile({loss: 'categoricalCrossentropy', optimizer: 'adam'});
```

Для выполнения многозначной многоклассовой классификации (в которой каждому примеру данных может соответствовать несколько из возможных классов) последовательность слоев должна завершаться плотным слоем с сигма-функцией

активации и количеством нейронов, равным числу всех возможных классов. В качестве функции потерь следует использовать `binaryCrossentropy`. Для целевых признаков должно использоваться k-битное федеративное кодирование (k-hot encoding):

```
const model = tf.sequential();
model.add(tf.layers.dense({units: 32, activation: 'relu', inputShape:
    [numInputFeatures]}));
model.add(tf.layers.dense({units: 32, activation: 'relu'}));
model.add(tf.layers.dense({units: numClasses: activation: 'sigmoid'}));
model.compile({loss: 'binaryCrossentropy', optimizer: 'adam'});
```

Для выполнения регрессии к вектору непрерывных значений, последовательность слоев должна завершаться плотным слоем с количеством нейронов, равным числу предсказываемых значений (часто это всего лишь одно число, скажем, цена объекта недвижимости или температура) и используемой по умолчанию линейной функцией активации. Для регрессии подходит несколько различных функций потерь, из которых чаще всего применяются `meanSquaredError` и `meanAbsoluteError`:

```
const model = tf.sequential();
model.add(tf.layers.dense({units: 32, activation: 'relu', inputShape:
    [numInputFeatures]}));
model.add(tf.layers.dense({units: 32, activation: 'relu'}));
model.add(tf.layers.dense({units: numClasses}));
model.compile({loss: 'meanSquaredError', optimizer: 'adam'});
```

Сверточные сети

Сверточные слои предназначены для поиска локальных пространственных закономерностей путем применения одного и того же геометрического преобразования к различным пространственным местоположениям (пятнам) во входном тензоре. В результате получаются трансляционно инвариантные представления, обеспечивающие эффективность работы с данными и модульность сверточных слоев. Эта идея применима к пространствам произвольной размерности: одномерным (последовательности), двумерным (изображения или аналогичное представление прочих сущностей, например спектрограммы звука), трехмерным (пространственные/объемные данные) и т. д. Для обработки последовательностей можно использовать слои tf.layers.conv1d, для изображений — слои conv2d, а для пространственных данных — conv3d.

Сверточные сети состоят из последовательностей сверточных слоев и слоев субдискретизации. С помощью слоев субдискретизации можно производить пространственную понижающую дискретизацию данных, необходимую для сохранения приемлемых размеров карт признаков при росте их (признаков) количества, а также для того, чтобы последующие слои могли «видеть» больший пространственный фрагмент входных изображений сверточной сети. Сверточные сети часто завершаются слоем схлопывания или глобальным слоем субдискретизации, преобразующими карты пространственных признаков в векторы, которые затем можно обработать с помощью последовательности плотных слоев (MLP) для получения результатов классификации или регрессии.

Весьма вероятно, что скоро обычную свертку повсеместно заменит эквивалентный, но более быстрый и эффективный алгоритм: разделяемая свертка с учетом глубины (слои `tf.layers.separableConv2d`). При создании сети с нуля очень рекомендуется использовать разделяемую свертку с учетом глубины. Слой separableConv2d можно использовать в качестве упрощенной замены `tf.layers.conv2d`, в результате чего получается меньшая и более быстрая сеть, демонстрирующая ничуть не худшие, а то и лучшие результаты. Ниже приведена типичная сеть для классификации изображений (однозначной многоклассовой классификации в данном случае). Ее топология включает повторяющиеся группы слоев свертки-субдискретизации:

```
const model = tf.sequential();
model.add(tf.layers.separableConv2d({
    filters: 32, kernelSize: 3, activation: 'relu',
    inputShape: [height, width, channels]}));
model.add(tf.layers.separableConv2d({
    filters: 64, kernelSize: 3, activation: 'relu'}));
model.add(tf.layers.maxPooling2d({poolSize: 2}));
model.add(tf.layers.separableConv2d({
    filters: 64, kernelSize: 3, activation: 'relu'}));
model.add(tf.layers.separableConv2d({
    filters: 128, kernelSize: 3, activation: 'relu'}));
model.add(tf.layers.maxPooling2d({poolSize: 2}));

model.add(tf.layers.separableConv2d({
    filters: 64, kernelSize: 3, activation: 'relu'}));
model.add(tf.layers.separableConv2d({
    filters: 128, kernelSize: 3, activation: 'relu'}));
model.add(tf.layers.globalAveragePooling2d());
model.add(tf.layers.dense({units: 32, activation: 'relu'}));
model.add(tf.layers.dense({units: numClasses, activation: 'softmax'}));

model.compile({loss: 'categoricalCrossentropy', optimizer: 'adam'});
```

Рекуррентные сети

RNN обрабатывают последовательности входных данных по одной метке времени за раз с сохранением состояния от начала и до конца. Состояние обычно представляет собой вектор или набор векторов (точку в геометрическом пространстве). В случае последовательностей, в которых нас интересуют не инвариантные относительно времени закономерности (например, данные временных рядов, в которых недавнее прошлое важнее, чем отдаленное).

В TensorFlow.js есть три типа слоев RNN: simpleRNN, GRU и LSTM. В большинстве случаев на практике применяются GRU или LSTM. LSTM, из этих двух типов, обладает большими возможностями, но и требует больших вычислительных ресурсов. GRU можно считать более простой и «дешевой» альтернативой LSTM.

Для размещения нескольких слоев RNN один поверх другого необходимо, чтобы каждый из них, за исключением последнего, возвращал полную последовательность выходных сигналов (каждый входной временной шаг соответствует выходному временному шагу). Если такого размещения не требуется, слой RNN обычно

возвращает только последний выходной сигнал, содержащий информацию обо всей последовательности.

Ниже приведен пример бинарной классификации последовательности векторов с помощью отдельного слоя RNN вместе с плотным слоем:

```
const model = tf.sequential();
model.add(tf.layers.lstm({
  units: 32,
  inputShape: [numTimesteps, numFeatures]
}));
model.add(tf.layers.dense({units: 1, activation: 'sigmoid'}));
model.compile({loss: 'binaryCrossentropy', optimizer: 'rmsprop'});
```

А вот модель с последовательностью слоев RNN для однозначной многоклассовой классификации последовательности векторов:

```
const model = tf.sequential();
model.add(tf.layers.lstm({
  units: 32,
  returnSequences: true,
  inputShape: [numTimesteps, numFeatures]
}));
model.add(tf.layers.lstm({units: 32, returnSequences: true}));
model.add(tf.layers.lstm({units: 32}));
model.add(tf.layers.dense({units: numClasses, activation: 'softmax'}));
model.compile({loss: 'categoricalCrossentropy', optimizer: 'rmsprop'});
```

Слои для сокращения переобучения и улучшения сходимости

Помимо вышеупомянутых основных типов слоев, существует несколько других вспомогательных типов слоев, применимых к широкому спектру моделей и задач. Без этих слоев достичь нынешних уровней безошибочности во многих задачах машинного обучения не удалось бы. Например, в MLP сверточные сети и RNN часто вставляют слои дропаута и нормализации по батчам для ускорения сходимости и сокращения переобучения. Следующий пример демонстрирует MLP для регрессии, содержащий слои дропаута:

```
const model = tf.sequential();
model.add(tf.layers.dense({
  units: 32,
  activation: 'relu',
  inputShape: [numFeatures]
}));
model.add(tf.layers.dropout({rate: 0.25}));
model.add(tf.layers.dense({units: 64, activation: 'relu'}));
model.add(tf.layers.dropout({rate: 0.25}));
model.add(tf.layers.dense({units: 64, activation: 'relu'}));
model.add(tf.layers.dropout({rate: 0.25}));
model.add(tf.layers.dense({
  units: numClasses,
  activation: 'categoricalCrossentropy'
}));
model.compile({loss: 'categoricalCrossentropy', optimizer: 'rmsprop'});
```

13.2.3. Использование предобученных моделей в TensorFlow.js

Если поставленная задача машинного обучения связана со спецификой конкретного приложения или набора данных, имеет смысл создать модель с нуля, и TensorFlow.js предоставляет все возможности для этого. Однако в некоторых случаях задача носит стандартный характер, и существуют предобученные модели, либо в точности соответствующие ее требованиям, либо требующие лишь небольших изменений. Имеется множество предобученных моделей как в TensorFlow.js, так и основанных на них моделей сторонних разработчиков с «чистыми» и удобными в использовании API. Существуют и соответствующие пакеты npm, которые удобно подключать в виде зависимостей в приложения JavaScript (включая веб-приложения и проекты Node.js).

Подобные предобученные модели могут заметно ускорить разработку в подходящих сценариях использования. Поскольку перечислить тут все предобученные модели на основе TensorFlow.js невозможно, рассмотрим только несколько наиболее популярных из известных нам. Пакеты с префиксом @tensorflow-models/ в названии созданы командой TensorFlow.js, а остальные — плоды труда сторонних разработчиков.

@tensorflow-models/mobilenet — облегченная модель классификации изображений, выдающая для входного изображения оценки вероятностей его принадлежности к 1000 классов ImageNet. Она удобна для маркирования изображений на веб-страницах и обнаружения заданного содержимого во входном потоке с веб-камеры, а также задач переноса обучения с изображениями на входе. Хотя @tensorflow-models/mobilenet ориентирована на общие классы изображений, существуют сторонние пакеты для предметно-ориентированной классификации изображений. Например, пакет nsfwjs классифицирует изображения на содержащие порнографический (и прочий неуместный) контент и допустимые, что удобно для родительского контроля, безопасного просмотра сайтов и тому подобных приложений.

Как мы обсуждали в главе 5, задача обнаружения объектов отличается от задачи классификации изображений, поскольку результаты должны включать информацию не только о том, *какие* объекты содержит изображение, но и *где* они располагаются в системе координат изображения. Модель обнаружения объектов @tensorflow-models/coco-ssd способна обнаруживать 90 классов объектов, причем может находить в каждом входном изображении при их наличии несколько целевых объектов, ограничивающие прямоугольники которых могут пересекаться (рис. 13.1, блок А).

Особый интерес для веб-приложений представляют определенные типы объектов, открывающие возможности для новых интересных видов взаимодействия человека с компьютером. В их числе человеческие лица, руки и тело в целом. Для каждого из этих трех типов объектов существуют специализированные сторонние модели на основе TensorFlow.js. Обнаружение лиц и частей лиц (например, глаз или рта; см. рис. 13.1, блок Б) в режиме реального времени поддерживают модели face-api.js и handsfree. Отслеживать местоположение одной или обеих рук в реальном времени может handtrackjs (см. рис. 13.1, блок В). Что касается тела в целом, обнаруживать с высокой точностью ключевые скелетные точки (плечи, локти, бедра и колени; см. рис. 13.1, блок Г) позволяет модель @tensorflow-models/posenet.

Д

text	Identity attack	Insult	obscene	severe toxicity	sexual explicit	threat	toxicity
We're dudes on cpmputers, moron. You are quite astonishingly stupid.	false	true	false	false	false	false	true
Please stop. If you continue to vandalize Wikipedia, as you did to Kmart, you will be blocked from editing.	false	false	false	false	false	false	false
I respect your point of view, and when this discussion originated on 8th April I would have tended to agree with you.	false	false	false	false	false	false	false

Рис. 13.1. Снимки экрана для нескольких предобученных моделей (существующих в виде пакетов npm), созданных с помощью TensorFlow.js. Блок A: @tensorflow-models/coco-ssd — многоцелевое средство обнаружения объектов. Блок B: модель face-api.js предназначена для распознавания лиц и их ключевых точек в режиме реального времени (воспроизводится из https://github.com/justadudewhohacks/face-api.js с разрешения Винсента Мюллера). Блок C: handtrack.js отслеживает местоположение одной или обеих рук в режиме реального времени (воспроизводится из https://github.com/victordibia/handtrack.js/ с разрешения Виктора Дибиа). Блок D: @tensorflow-models/posenet распознает ключевые точки тела человека в реальном времени на основе входных изображений. Блок E: @tensorflow-models/toxicity обнаруживает и маркирует семь типов недопустимого содержимого в любом входном тексте на английском языке

Что касается входных аудиоданных, предобученная модель @tensorflow-models/speech-commands способна распознавать в режиме реального времени 18 английских слов с помощью непосредственно API WebAudio браузера. И хотя ее возможности не так широки, как у средств распознавания речи с большим словарем, она позволяет пользователям взаимодействовать в браузере с программой голосом.

Существуют также предобученные модели для текстовых входных данных. Например, модель @tensorflow-models/toxicity определяет степень деструктивности заданного английского входного текста по нескольким направлениям (например, угрозы, оскорбления или непристойности), что весьма полезно для полуавтоматического

модерирования контента (см. рис. 13.1, блок Д). Модель toxicity основана на более универсальной модели обработки естественного языка — @tensorflow-models/universal-sentence-encoder, — ставящей в соответствие любому заданному предложению на английском языке вектор, который далее можно использовать при решении широкого спектра задач обработки естественного языка, например классификации намерений, классификации по темам, анализа тональностей и формирования ответов на вопросы.

Следует подчеркнуть, что некоторые из упомянутых моделей не только поддерживают простое выполнение вывода, но и могут служить основой для переноса обучения или дальнейшего машинного обучения, позволяя тем самым применять их к предметно-ориентированным данным, без длительного процесса создания или обучения модели. А все благодаря сочетаемости, подобно кубикам LEGO, слоев и моделей. Например, выходной сигнал универсального кодировщика предложений предназначен в основном для использования в расположенных далее по конвейеру моделей. В модель speech-commands встроена поддержка сбора голосовых сэмплов для новых классов слов и обучения на этих сэмплах нового классификатора, что удобно для приложений голосового управления, требующих пользовательского словаря или приспособления к голосу конкретного пользователя. Кроме того, выходные сигналы от таких моделей, как PoseNet и face-api.js, относительно местоположения в текущий момент головы, рук или позы тела можно подавать на вход последующей модели распознавания конкретных жестов или последовательностей движений, что удобно для многих приложений, например альтернативных средств общения для лиц с ограниченными возможностями.

Помимо вышеупомянутых типов входных данных, существуют также сторонние предобученные модели на основе TensorFlow.js для художественного творчества. Например, ml5.js включает модель для быстрого переноса стиля изображений и модель для автоматического рисования эскизов. @magenta/music включает модель для записи в виде нот фортепианной музыки (преобразования звука в ноты) и MusicRNN (языковую модель для мелодий), способные «сочинять» музыку на основе нескольких начальных нот, помимо других захватывающих предобученных моделей.

Набор предобученных моделей очень велик и продолжает расширяться. Сообщества JavaScript-разработчиков и специалистов по глубокому обучению отличаются открытостью и духом общего соучастия. При дальнейшем путешествии в мир глубокого обучения, возможно, вы наткнетесь на новые интересные идеи, которые могут пригодиться другим разработчикам. Так что смелее, обучайте, превращайте в пакеты, загружайте свои модели в npm в форме предобученных моделей и улучшайте их в соответствии с обратной связью от пользователей. Тогда вы станете полноценным участником сообщества глубокого обучения на JavaScript.

13.2.4. Спектр возможностей

Какие полезные и интересные модели можно создать, используя все эти слои и предобученные модели в качестве «кирпичиков»? Помните, создание моделей глубокого обучения подобно кубикам LEGO: слои и модули можно комбинировать для отображения практически чего угодно во что угодно, лишь бы можно было представить

входные и выходные данные в виде тензоров, и формы входных/выходных тензоров слоев были совместимы. Полученная последовательность слоев — модель — производит дифференцируемое геометрическое преобразование, способное обучиться отображать входные данные в выходные, лишь бы их взаимосвязь не была слишком сложной относительно разрешающих возможностей модели. При такой парадигме спектр возможностей неограничен. В этом разделе мы приведем несколько примеров, дабы вдохновить вас выйти за рамки простейших задач классификации и регрессии, которым мы уделяли основное внимание в этой книге.

Мы отсортировали варианты по типам входных и выходных данных. Учтите, что довольно многие из них слегка выходят за пределы возможного. И хотя при достаточном объеме обучающих данных можно обучить модель для решения любой из этих задач, она вряд ли будет хорошо обобщаться на данные, отличные от обучающих.

- Отображение вектора в вектор.
  - *Прогнозы в сфере здравоохранения* (Predictive healthcare) — отображение медицинских карт пациентов в предсказанные результаты лечения.
  - *Поведенческий таргетинг* (Behavioral targeting) — отображение множества атрибутов веб-сайта в поведение пользователей на этом сайте (включая просмотры страниц, щелчки на ссылках и другие действия).
  - *Контроль качества продукции* (Product quality control) — отображение множества атрибутов произведенного товара в прогноз его успешности на рынке (продаж и доходов в различных областях рынка).
- Отображение изображения в вектор.
  - *ИИ для медицинских снимков* (Medical image AI) — отображение медицинских снимков (например, рентгеновских) в результаты обследования.
  - *Автоматическое управление автомобилем* (Automatic vehicle steering) — отображение изображений с камер автомобиля в сигналы управления им, например поворот руля.
  - *Виртуальный диетолог* (Diet helper) — отображение изображений еды и различных блюд в прогноз их влияния на здоровье (например, предсказание количества калорий или предупреждение о возможной аллергической реакции).
  - *Рекомендации косметики* (Cosmetic product recommendation) — на основании селфи пользователя предоставление рекомендаций по косметике.
- Отображение данных временных рядов в вектор.
  - *Нейрокомпьютерные интерфейсы* (Brain-computer interfaces) — отображение сигналов электроэнцефалограмм (ЭЭГ) в намерения пользователя.
  - *Поведенческий таргетинг* — отображение истории покупок (например, книг или фильмов) в вероятности покупки других товаров в будущем.
  - *Предсказание землетрясений и повторных толчков* — отображение последовательностей данных с сейсмографов в прогнозы вероятности землетрясений и повторных толчков.

- Отображение текста в вектор.
  - *Сортировка сообщений электронной почты* (E-mail sorter) — отображение содержимого сообщения электронной почты в универсальные или задаваемые пользователем метки (например, «работа», «семья» или «спам»).
  - *Оценка грамматики* (Grammar scorer) — отображение сочинений студентов в оценки правильности грамматики.
  - *Сортировка больных* (Speech-based medical triaging) — на основании жалоб пациента рекомендация, в какое отделение больницы ему следует обратиться.
- Отображение текста в текст.
  - *Рекомендации ответных сообщений* (Reply-message suggestion) — отображение сообщений электронной почты в множество возможных ответных сообщений.
  - *Формирование предметно-ориентированных ответов на вопросы* (Domain-specific question answering) — отображение вопросов пользователя в тексты автоматических ответов.
  - *Автоматическое реферирование* (Summarization) — отображение длинной статьи в короткое резюме.
- Отображение изображений в текст.
  - *Автоматическая генерация замещающего текста* (Automated alt-text generation) — генерация по входному изображению короткого фрагмента текста, отражающего саму сущность содержимого этого изображения.
  - *Виртуальный поводырь для слабовидящих* (Mobility aids for the visually impaired) — отображение изображений окружающей обстановки внутри или вне здания в голосовые инструкции и предупреждения относительно возможных опасностей (например, местоположение выходов и различных препятствий).
- Отображение изображений в изображения.
  - *Повышение разрешения изображений* (Image super-resolution) — отображение изображений в низком разрешении в изображения в более высоком.
  - *Восстановление трехмерных изображений* (Image-based 3D reconstruction) — отображение обычных изображений в изображения того же объекта, но с другого ракурса.
- Отображение изображений и данных временных рядов в вектор.
  - *Виртуальный помощник доктора для различных типов входных данных* (Doctor's multimodal assistant) — отображение медицинских снимков пациентов (например, МРТ) и истории показателей жизнедеятельности (артериальное давление, частота сердечных сокращений и т. д.) в прогноз результатов лечения.
- Отображение изображения и текста в текст.
  - *Формирование ответов на вопросы на основе изображений* (Image-based question answering) — отображение изображения и относящегося к нему вопроса (например, изображения подержанного автомобиля и вопроса о его марке и годе выпуска) в ответ.

- Отображение изображений и вектора в изображение.
  - *Виртуальная примерка одежды и косметики* (Virtual try-on for clothes and cosmetic products) — отображение селфи пользователя и векторного представления косметического средства или предмета одежды в изображение пользователя, на которого надета данная одежда или нанесена косметика.
- Отображение данных временного ряда и вектора в данные временного ряда.
  - *Перенос музыкального стиля* (Musical style transfer) — отображение партитуры (например, произведения классической музыки в виде временного ряда нот) и описания желаемого состояния (например, «джаз») в новую партитуру, написанную в этом стиле.

Как вы могли заметить, последние четыре категории в этом списке включают смешанные типы входных данных. На современном этапе развития технологии, когда все вокруг оцифровывается, а потому может быть представлено в виде тензоров, возможности глубокого обучения ограничиваются лишь вашим воображением и наличием обучающих данных. И хотя возможны почти все виды отображений, но не все. Мы обсудим в следующем разделе, на что глубокое обучение *не* способно.

13.2.5. Ограничения глубокого обучения

Спектр приложений, которые можно реализовать с помощью глубокого обучения, практически неисчерпаем. В результате очень легко переоценить мощь глубоких нейронных сетей и чересчур оптимистично отнестись к перечню задач, которые они могут решить. В этом разделе мы вкратце поговорим о некоторых текущих их ограничениях.

Нейронные сети видят мир не так, как люди

Пытаясь разобраться в глубоком обучении, важно не слишком *очеловечивать* его — существует тенденция неправильно интерпретировать глубокие нейронные сети как некую имитацию восприятия и мышления людей. Очеловечивать глубокие нейронные сети, вне всякого сомнения, будет неправильно в нескольких отношениях. Во-первых, воспринимая внешний стимул (например, изображение лица девушки или изображение зубной щетки), человек не только воспринимает яркость и цветовые закономерности входного сигнала, но и выделяет из него более глубокие и важные концепты, отражаемые этими поверхностными закономерностями (например, лицо молодой девушки или товар, связанный с гигиеной ротовой полости, а также связи между ними). Глубокие нейронные сети работают не так. При обучении модели, предназначенной для добавления подписей к изображениям, отображать изображения в текстовый выходной сигнал не следует считать, что модель «понимает» изображение подобно человеку. В некоторых случаях модель может начать генерировать абсурдные подписи даже при незначительном отклонении от представленной в обучающих данных разновидности изображений (см. рис. 13.3).

В частности, своеобразный, нечеловеческий способ обработки глубокими нейронными сетями входных данных иллюстрируется *состязательными примерами*

данных (adversarial examples), специально создаваемыми для того, чтобы обманом заставить модель глубокого обучения допускать ошибки классификации. Мы показали при поиске наиболее активирующих изображений для фильтров сверточной сети в разделе 7.2, как производить градиентный подъем в пространстве входных данных с целью максимизации активации фильтра сверточной сети. Эту идею можно распространить на выходные вероятности и максимизировать предсказываемые моделью вероятности для любого выходного класса с помощью градиентного подъема в пространстве входных данных. Мы можем добавить к изображению панды градиент гиббона и модель ошибочно классифицирует данное изображение как изображение гиббона (рис. 13.3), несмотря на то что градиент гиббона мал по величине и напоминает шум, так что получившееся в результате состязательное изображение не отличается для человеческого глаза от исходного изображения панды.

«Мальчик держит бейсбольную биту».

Рис. 13.2. Ошибка модели, предназначенной для добавления подписей к изображениям, обученной с помощью глубокого обучения

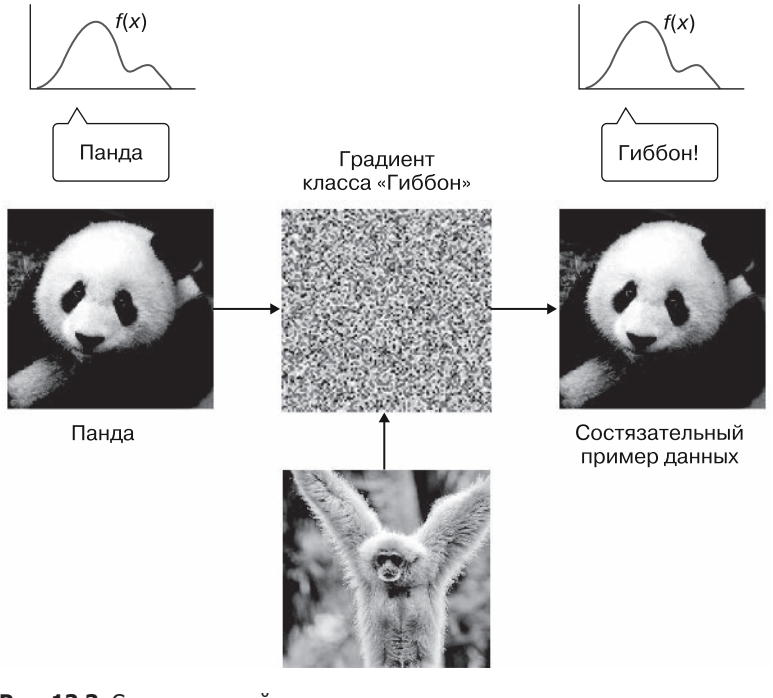


Рис. 13.3. Состязательный пример данных: незаметные человеческому глазу изменения могут привести к неправильному результату классификации глубокой сверточной сетью. См. дальнейшее обсуждение состязательных атак на глубокие нейронные сети по адресу http://mng.bz/pyGz

Таким образом, предназначенные для компьютерного зрения глубокие нейронные сети не способны «понимать» изображения, по крайней мере в общечеловеческом смысле. Еще одна область, в которой обучение людей резко отличается от глубокого обучения, — обобщение на данные, выходящие за рамки ограниченного количества обучающих примеров. Глубокие нейронные сети способны на так называемое *локальное обобщение* (local generalization). На рис. 13.4 приведен сценарий, в котором перед глубокой нейронной сетью и человеком ставится одна и та же задача усвоения границ отдельного класса в двумерном параметрическом пространстве исходя из очень небольшого количества (скажем, восьми) обучающих примеров данных. Человек понимает, что граница класса должна быть гладкой, а область — связной, и быстро интуитивно рисует замкнутую кривую. Нейронной сети, в отличие от него, недостает абстракции и априорных знаний, в результате чего она может нарисовать несколько импровизированных неровных границ, сильно подогнанных к немногочисленным имеющимся обучающим примерам данных. Обученная на этих примерах модель будет очень плохо обобщаться на другие данные. Помочь может добавление дополнительных примеров данных, но это не всегда возможно на практике. Основная проблема: нейронная сеть создается с нуля под конкретную задачу. В отличие от человека, у нее нет никаких априорных знаний, поэтому она не знает, «чего ждать»[1]. В этом и состоит основная причина главного ограничения современных алгоритмов глубокого обучения, а именно: для достижения глубокой нейронной сетью удовлетворительной точности обобщения необходим большой объем маркированных вручную обучающих данных.

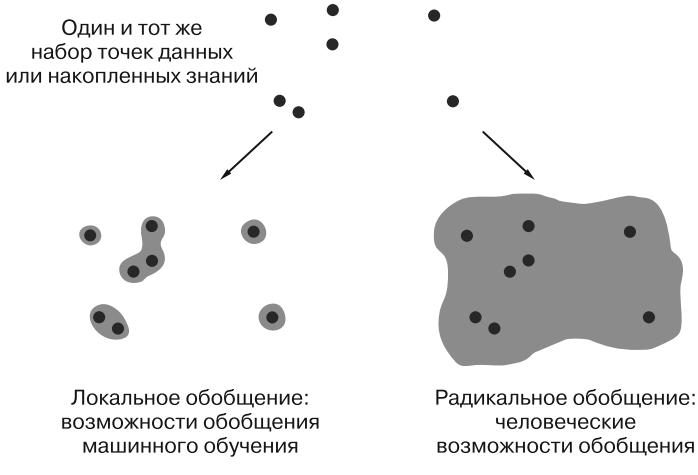


Рис. 13.4. Локальное обобщение в моделях глубокого обучения и радикальное обобщение человеческого интеллекта

[1] Исследователи пробовали обучать одну глубокую нейронную сеть на множестве различных и, казалось бы, не связанных между собой задач ради совместного использования знаний из различных областей (см., например: *Kaiser L. et al.* One Model To Learn Them All // submitted 16 Jun. 2017. https://arxiv.org/abs/1706.05137). Но подобные многоцелевые модели пока широкого распространения не получили.

13.3. Современные тенденции глубокого обучения

Как мы уже говорили, прогресс глубокого обучения за последние годы был поистине впечатляющим, но некоторые ограничения все еще ему присущи. Но эта сфера знаний не стоит на месте, а эволюционирует с поразительной быстротой, так что вполне возможно, что в ближайшем будущем некоторые из этих ограничений будут устранены. В этом разделе приведен набор обоснованных прогнозов возможных технических прорывов в глубоком обучении на ближайшие годы.

- Прежде всего можно ожидать серьезного прогресса в обучении без учителя и обучении с учителем, что, в свою очередь, самым кардинальным образом повлияет на все формы глубокого обучения, ведь, в то время как найти или создать маркированный набор данных непросто, немаркированных наборов данных во всех предметных областях полным-полно. Способ направлять обучение на большом массиве немаркированных данных с помощью небольшого объема маркированных открыл бы дорогу к множеству новых приложений глубокого обучения.
- Во-вторых, подходящее для глубокого обучения аппаратное обеспечение также будет развиваться и будут появляться все более и более мощные средства ускорения для нейронных сетей (например, будущие поколения тензорных процессоров, TPU[1]), что позволит исследователям обучать обладающие все большими возможностями нейронные сети на еще больших наборах данных, тем самым увеличивая степень безошибочности во многих задачах машинного обучения, в частности в компьютерном зрении, распознавании речи, обработке естественного языка и генеративных моделях.
- Проектирование архитектуры моделей и подбор гиперпараметров моделей, вероятно, будут все более и более автоматизироваться. Мы уже видим четкие тенденции в этом направлении, что демонстрируют такие технологии, как AutoML[2] и Google Vizier[3].
- Совместное использование и переиспользование компонентов нейронных сетей также будет расширяться. Равно как и масштабы переноса обучения на основе предобученных моделей. Современные модели глубокого обучения с каждым днем становятся все более мощными и универсальными. Их обучают на все больших и больших наборах данных, иногда с колоссальным расходом вычислительных ресурсов ради автоматизации проектирования архитектуры и подбора

---

1 *Jouppi N. P. et al.* In-Datacenter Performance Analysis of a Tensor Processing Unit™, 2017. https://arxiv.org/pdf/1704.04760.pdf.

2 *Zoph B., Le Q. V.* Neural Architecture Search with Reinforcement Learning // submitted 5 Nov. 2016. https://arxiv.org/abs/1611.01578.

3 *Golovin D.* Google Vizier: A Service for Black-Box Optimization // Proc. 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2017. Pp. 1487–1495. http://mng.bz/O9yE.

гиперпараметров (см. первое и второе наши предсказания). В результате все больший смысл, в том числе с экономической точки зрения, будет переиспользовать подобные модели либо непосредственно для выполнения вывода, либо для переноса обучения, вместо того чтобы обучать их с нуля. В некотором смысле это уподобляет сферу глубокого обучения проектированию обычного программного обеспечения, зависящего от высококачественных библиотек и регулярно переиспользующего их ради ускорения разработки и стандартизации всей сферы в целом.

- Возникнут новые сферы применения глубокого обучения, совершенствуя существующие решения и открывая возможности новых сценариев использования на практике. По нашему мнению, возможные сферы применения его практически безграничны. Такие области, как сельское хозяйство, финансы, образование, перевозки, здравоохранение, мода, спорт и развлечения, открывают бесчисленные возможности для пытливых специалистов по глубокому обучению.
- По мере проникновения глубокого обучения во все новые предметные области, вероятно, все заметнее будет акцент на оконечные устройства, поскольку именно они ближе всего к пользователям. В результате, вероятно, будут появляться все меньшие и более энергоэффективные архитектуры нейронных сетей с такой же степенью безошибочности, что и существующие ныне большие.

Все эти предсказания окажут свое влияние и на глубокое обучение на JavaScript, особенно последние три. Так что ждите новых, более мощных и эффективных моделей в TensorFlow.js в будущем.

13.4. Рекомендации по дальнейшему изучению

В качестве заключительных слов хотелось бы дать вам небольшие рекомендации о том, как продолжать изучение и поддерживать актуальность своих знаний и навыков после завершения чтения этой книги. Сфере современного глубокого обучения в нынешнем виде всего несколько лет, несмотря на его предысторию, медленно разворачивавшуюся на протяжении нескольких десятилетий. Благодаря стремительному росту финансовых ресурсов и количества исследователей начиная с 2013 года вся эта сфера ныне развивается в бешеном темпе. Многое из материала этой книги достаточно быстро устареет, но основные идеи глубокого обучения (обучение на данных, сокращение объема ручного проектирования признаков, послойные преобразования представлений), вероятно, окажутся долговечнее. Но главное, что на основе полученного при чтении этой книги фундамента знаний, надеемся, вы сможете самостоятельно продолжить изучение новых технологий и тенденций в сфере глубокого обучения. К счастью, эта область отличается открытостью и большинство самых современных достижений (включая многие наборы данных!) публикуется в виде свободно и бесплатно доступных препринтов, сопровождаемых общедоступными сообщениями в блогах и твитами. Далее рассмотрены несколько основных ресурсов, которые могут вам пригодиться.

13.4.1. Отрабатывайте навыки решения реальных задач на Kaggle

Эффективный способ получить реальный опыт машинного обучения (и особенно глубокого обучения) — попробовать свои силы в конкурсах Kaggle (https://kaggle.com/). По-настоящему разобраться в машинном обучении можно только посредством написания кода, создания моделей и их настройки. Это и есть главная идея данной книги, как видно из многочисленных примеров кода, только и ждущих изучения, доработки и использования в качестве основы для ваших программ. Но самый эффективный способ изучить МО — создавать свои собственные модели и системы машинного обучения с нуля с помощью библиотек наподобие TensorFlow.js. На сайте Kaggle можно найти постоянно обновляемый перечень конкурсов по обработке данных и наборов данных, многие из которых связаны с глубоким обучением.

И хотя большинство пользователей Kaggle применяют в конкурсах инструменты для языка Python (например, TensorFlow и Keras), большинство наборов данных на Kaggle от языка программирования не зависит. Поэтому вполне реально решать большинство задач Kaggle с помощью фреймворков глубокого обучения отличных от Python языков программирования, например TensorFlow.js. Поучаствовав в нескольких конкурсах, возможно, в составе команды разработчиков вы познакомитесь с практической стороной описанных в этой книге рекомендуемых передовых практик, особенно подбора гиперпараметров и предотвращения переобучения на проверочном наборе данных.

13.4.2. Читайте о последних новинках на arXiv

Исследования в сфере глубокого обучения, в отличие от некоторых других научных областей, происходят практически полностью на виду. Статьи открываются для всеобщего (бесплатного) доступа сразу же после завершения и рецензирования, а огромное количество соответствующего программного обеспечения отличается открытым исходным кодом. ArXiv (https://arxiv.org/) — произносится «а(р)хив» (X — это греческая буква «хи») — сервер препринтов со свободным доступом для научных статей по математике, физике и теории вычислительной техники. Он стал фактически способом по умолчанию публикации передовых работ в области машинного обучения и глубокого обучения, а потому — способом по умолчанию для поддержания актуальности своих знаний в этой области. Благодаря этому данная сфера знаний развивается исключительно быстро: все новые открытия и изобретения открыты для всеобщего обозрения, критических замечаний и дальнейшего развития.

Существенный недостаток ArXiv — само количество публикуемых ежедневно новых статей, так что просмотреть их все невозможно. А поскольку многие из статей в ArXiv — нерецензированные, найти наиболее важные и качественные непросто. К счастью, были созданы инструменты для решения этой проблемы. Например, с помощью веб-сайта ArXiv Sanity Preserver[1] (arxiv-sanity.com) — системы рекомендации

[1] Буквально «хранитель душевного равновесия для ArXiv». — *Примеч. пер.*

новых статей ArXiv — можно следить за новинками в конкретных вертикальных предметных областях глубокого обучения (например, в обработке естественного языка или обнаружении объектов). Кроме того, вы можете следить за публикациями в интересующих вас областях или ваших излюбленных авторов с помощью Google Scholar.

13.4.3. Исследование экосистемы TensorFlow.js

TensorFlow.js обладает динамичной и быстро растущей экосистемой, состоящей из документации, различных руководств, учебников, блогов и проектов с открытым исходным кодом.

- Основной источник информации при работе с TensorFlow.js — официальная онлайн-документация по адресу https://www.tensorflow.org/js/. Подробную актуальную документацию можно найти по адресу https://js.tensorflow.org/api/latest/.
- Вопросы относительно TensorFlow.js можно задавать на сайте Stack Overflow по тегу *tensorflow.js*: https://stackoverflow.com/questions/tagged/tensorflow.js.
- Общее обсуждение данной библиотеки можно найти в группе Google: https://groups.google.com/a/tensorflow.org/forum/#!forum/tfjs.
- Можете также подписаться на членов команды TensorFlow.js, активно пишущих в Twitter, включая:
  - https://twitter.com/sqcai;
  - https://twitter.com/nsthorat;
  - https://twitter.com/dsmilkov;
  - https://twitter.com/tensorflow.

Заключительные слова

Вот вы и добрались до конца книги! Надеемся, вы узнали кое-что про ИИ, глубокое обучение и решение простых задач глубокого обучения на JavaScript с помощью TensorFlow.js. Изучение ИИ и глубокого обучения, как и любой интересный и полезный предмет, — занятие на всю жизнь. То же самое относится и к приложению ИИ и глубокого обучения к практическим задачам. Это справедливо как для профессионалов, так и для любителей. Несмотря на все текущие достижения в сфере глубокого обучения, ответов на большинство фундаментальных его вопросов все еще не найдено и большая часть реального его потенциала еще не раскрыта. Изучайте, задавайте вопросы, исследуйте, мечтайте, влезайте в тонкости, создавайте и делитесь! Нам не терпится увидеть, что вы создадите с помощью глубокого обучения и JavaScript!

Приложения

A Установка библиотеки tfjs-node-gpu и ее зависимостей

Для использования версии TensorFlow.js с GPU-ускорением (tfjs-node-gpu) на машине должны быть установлены библиотеки CUDA и CuDNN. Прежде всего машина должна быть оснащена GPU производства NVIDIA с поддержкой CUDA. Чтобы проверить, удовлетворяет ли GPU вашей машины этому требованию, зайдите на сайт https://developer.nvidia.com/cuda-gpus.

Далее мы перечислим подробные шаги по установке драйвера и библиотеки для Linux и Windows — двух операционных систем, поддерживающих в настоящее время tfjs-node-gpu.

A.1. Установка tfjs-node-gpu в Linux

1. Мы предполагаем, что в вашей системе уже установлены Node.js и npm, а пути к node и npm включены в системный путь. Если нет — скачать установочные пакеты вы можете по адресу https://nodejs.org/en/download/.
2. Скачайте CUDA Toolkit по адресу https://developer.nvidia.com/cuda-downloads. Выберите версию, соответствующую желаемой версии tfjs-node-gpu. На момент написания данной книги последняя версия tfjs-node-gpu — 1.2.10, совместимая с CUDA Toolkit 10.0. Кроме того, не забудьте выбрать нужную операционную систему (Linux), архитектуру (например, x86_64 для машин с наиболее распространенными CPU Intel), дистрибутив Linux и его версию. Вам будет предложено скачать один из нескольких вариантов установочных пакетов. В последующих шагах предполагается, что вы скачали файл `runfile (local)` (а не, например, локальный пакет `.deb`).

3. В каталоге, куда вы скачали упомянутый файл, выполните команду `chmod`, чтобы сделать его исполняемым. Например:

```
chmod +x cuda_10.0.130_410.48_linux.run
```

4. Запустите этот файл с помощью команды `sudo`. Учтите, что установочному процессу CUDA Toolkit может понадобиться установить или обновить драйвер NVIDIA на вашей машине, если его установленная версия устарела или ее вообще нет. В этом случае необходимо остановить сервер X, перейдя в режим использования только командной оболочки. В дистрибутивах Ubuntu и Debian войти в консоль можно с помощью сочетания клавиш Ctrl+Alt+F1.

   Следуйте подсказкам на экране для установки CUDA Toolkit, после чего машина будет перезагружена. Если вы работали в режиме командной оболочки, можете после перезагрузки вернуться в обычный режим GUI.

5. В случае успешного завершения шага 3 системный путь будет включать команду `nvidia-smi`, с помощью которой можно проверить состояние GPU. Она выводит, помимо версии текущего драйвера NVIDIA, такую информацию, как названия, показания датчиков температуры, скорость вентиляторов, а также коэффициенты загрузки процессоров и памяти установленных в системе GPU. Это удобная утилита для мониторинга GPU в режиме реального времени при использовании tfjs-node-gpu для обучения глубоких нейронных сетей. Типичное сообщение, выводимое командой `nvidia-smi`, выглядит следующим образом (для машин с двумя GPU NVIDIA):

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.111                 Driver Version: 384.111                  |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Quadro P1000        Off  | 00000000:65:00.0  On |                  N/A |
| 41%   53C    P0   ERR! /  N/A |    620MiB /  4035MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Quadro M4000        Off  | 00000000:B3:00.0 Off |                  N/A |
| 46%   30C    P8    11W / 120W |      2MiB /  8121MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      3876      G   /usr/lib/xorg/Xorg                           283MiB |
```

6. Добавьте путь к 64-битным файлам библиотеки CUDA в переменную среды `LD_LIBRARY_PATH`. В случае использования командной оболочки bash можно добавить следующую строку в файл `.bashrc`:

```
export LD_LIBRARY_PATH="/usr/local/cuda/lib64:${PATH}"
```

   tfjs-node-gpu ищет необходимые для запуска динамические файлы библиотеки по путям из переменной среды `LD_LIBRARY_PATH`.

7. Скачайте CuDNN с сайта https://developer.nvidia.com/cudnn. Зачем нужно еще и CuDNN, помимо CUDA? Потому, что CUDA — универсальная вычислительная библиотека, применяемая и в других сферах, помимо глубокого обучения (например, в гидродинамике). CuDNN — основанная на CUDA библиотека NVIDIA для ускорения операций с глубокими нейронными сетями.

- Для скачивания CuDNN вам может понадобиться создать учетную запись NVIDIA и ответить на несколько вопросов анкеты. Убедитесь, что скачали версию CuDNN, соответствующую установленной на предыдущих шагах версии CUDA Toolkit. Например, CuDNN 7.6 подходит для совместного использования с CUDA Toolkit 10.0.

8. В отличие от CUDA Toolkit, CuDNN поставляется не в виде исполняемого файла средства установки, а в виде сжатого архива tar, содержащего несколько файлов динамических библиотек и заголовков C/C++. Необходимо извлечь эти файлы и скопировать их по соответствующему пути установки. Для этого можно использовать последовательность команд наподобие следующей:

```
tar xzvf cudnn-10.0-linux-x64-v7.6.4.38.tgz
cp cuda/lib64/* /usr/local/cuda/lib64
cp cuda/include/* /usr/local/cuda/include
```

9. Теперь, после установки всех требуемых драйверов и библиотек, можно быстро проверить CUDA и CuDNN, импортировав tfjs-node-gpu в node:

```
npm i @tensorflow/tfjs @tensorflow/tfjs-node-gpu
node
```

Далее в интерфейсе командной строки Node.js вы увидите:

```
> const tf = require('@tensorflow/tfjs');
> require('@tensorflow/tfjs-node-gpu');
```

В случае успеха вы увидите несколько строк журнального вывода, подтверждающих, что GPU (или несколько GPU, в зависимости от конфигурации вашей системы) обнаружен (-ы) и tfjs-node-gpu может их использовать:

```
2018-09-04 13:08:17.602543: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1405] Found device 0
with properties:
name: Quadro M4000 major: 5 minor: 2 memoryClockRate(GHz): 0.7725
pciBusID: 0000:b3:00.0
totalMemory: 7.93GiB freeMemory: 7.86GiB
2018-09-04 13:08:17.602571: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1484] Adding visible
gpu devices: 0
2018-09-04 13:08:18.157029: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:965] Device
interconnect StreamExecutor with strength 1 edge matrix:
2018-09-04 13:08:18.157054: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:971]      0
2018-09-04 13:08:18.157061: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:984] 0:   N
2018-09-04 13:08:18.157213: I
```

```
tensorflow/core/common_runtime/gpu/gpu_device.cc:1097] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with
7584 MB memory) -> physical GPU (device: 0, name: Quadro M4000, pci bus
id: 0000:b3:00.0, compute capability: 5.2)
```

10. Все готово для использования всех возможностей tfjs-node-gpu. Достаточно включить в `package.json` следующие зависимости (или последующие их версии):

```
...
"dependencies": {
  "@tensorflow/tfjs": "^0.12.6",
  "@tensorflow/tfjs-node": "^0.1.14",
  ...
}
...
```

Импортируйте в файле `main.js` основные зависимости, включая `@tensorflow/tfjs` и `@tensorflow/tfjs-node-gpu`. Первая из них содержит общий API TensorFlow.js, а вторая подключает операции TensorFlow.js к быстродействующим вычислительным ядрам, реализованным на CUDA и CuDNN:

```
const tf = require('@tensorflow/tfjs');
require('@tensorflow/tfjs-node-gpu');
```

A.2. Установка tfjs-node-gpu в Windows

1. Убедитесь, что ваша версия Windows удовлетворяет системным требованиям CUDA Toolkit. CUDA Toolkit не поддерживает некоторые выпуски Windows и 32-битные архитектуры вычислительных систем. См. подробности по адресу https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html#system-requirements.
2. Мы предполагаем, что в вашей системе уже установлены Node.js и npm, а пути к node и npm включены в системную переменную `Path`. Если нет — скачать установочные пакеты вы можете по адресу https://nodejs.org/en/download/.
3. Установите Microsoft Visual Studio — она нужна для установки CUDA Toolkit. Какую версию Microsoft Visual Studio выбрать, можно узнать по той же ссылке, что и в шаге 1.
4. Скачайте и установите CUDA Toolkit для Windows. На момент написания данной книги для работы tfjs-node-gpu (текущая версия 1.2.10) требуется CUDA 10.0. Выберите соответствующий вашей версии Windows установочный пакет. Доступны установочные пакеты для Windows 7 и Windows 10. Этот шаг требует прав администратора.
5. Скачайте CuDNN. Убедитесь, что версия CuDNN соответствует версии CUDA. Например, CuDNN 7.6 подходит для совместного использования с CUDA Toolkit 10.0. Прежде чем скачать CuDNN, вам может понадобиться создать учетную запись NVIDIA и ответить на несколько вопросов анкеты.

6. В отличие от установочного пакета CUDA Toolkit, CuDNN поставляется в виде не исполняемого установочного файла, а файла архива ZIP. Распакуйте его, и вы увидите внутри три каталога: `cuda/bin`, `cuda/include` и `cuda/lib/x64`. Найдите каталог установки CUDA Toolkit (по умолчанию что-то вроде `C:/Program Files/NVIDIA CUDA Toolkit 10.0/cuda`). Скопируйте извлеченные файлы в соответствующие подкаталоги с теми же названиями. Например, извлеченные из архива файлы из каталога `cuda/bin` необходимо скопировать в `C:/Program Files/NVIDIA CUDA Toolkit 10.0/cuda/bin`. Для этого шага могут также потребоваться права администратора.
7. После установки CUDA Toolkit и CuDNN перезагрузите операционную систему Windows. Мы выяснили, что это необходимо для правильной загрузки (для использования в tfjs-node-gpu) всех только что установленных библиотек.
8. Установите пакет npm `window-build-tools`, необходимый для установки пакета npm `@tensorflow/tfjs-node-gpu` на следующем шаге:

   ```
   npm install --add-python-to-path='true' --global windows-build-tools
   ```

9. Установите с помощью npm пакеты `@tensorflow/tfjs` и `@tensorflow/tfjs-node-gpu`:

   ```
   npm -i @tensorflow/tfjs @tensorflow/tfjs-node-gpu
   ```

10. Для проверки успешности установки откройте командную строку node и выполните:

    ```
    > const tf = require('@tensorflow/tfjs');
    > require('@tensorflow/tfjs-node-gpu');
    ```

Убедитесь, что обе команды завершились без ошибок. После второй команды вы увидите несколько строк журналов, выведенных разделяемой библиотекой GPU TensorFlow. В них перечисляются GPU с поддержкой CUDA, которые были обнаружены tfjs-node-gpu и будут использоваться в последующих программах глубокого обучения.

Б *Краткое руководство по тензорам и операциям над ними в TensorFlow.js*

Это приложение посвящено прочим, не относящимся к `tf.Model`, частям API TensorFlow.js. И хотя `tf.Model` содержит полный набор методов, необходимых для обучения/оценки моделей и для получения на их основе дальнейшего вывода, зачастую для работы с объектами `tf.Model` необходимы прочие части TensorFlow.js. Основные сценарии:

- преобразование данных в тензоры для передачи в объекты `tf.Model`;
- выдача выполняемых `tf.Model` предсказаний (изначально находящихся в формате тензоров) в виде, подходящем для использования прочими частями программы.

Как вы увидите, вставка данных в тензоры и извлечение данных оттуда не представляет сложностей, но имеет смысл отметить некоторые общепринятые шаблоны действий и нюансы.

Б.1. Создание тензоров и соглашения по поводу их осей координат

Не забывайте, что *тензор* — просто контейнер для данных. У любого тензора есть два фундаментальных свойства: тип данных (dtype) и форма. *dtype* определяет, какие виды значений можно хранить в данном тензоре. Конкретный тензор способен хранить ровно один вид значений. На момент написания данной книги (версия 0.13.5) в TensorFlow.js поддерживались следующие dtype: float32, int32 и bool.

Форма (shape) — это массив целых чисел, определяющий количество элементов тензора и то, как они организованы. Ее можно считать «формой и размером» контейнера-тензора (рис. Б.1).

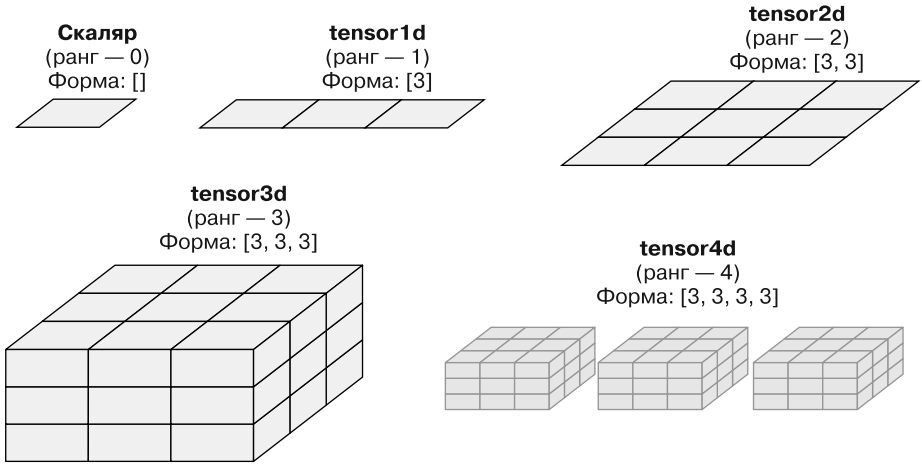


Рис. Б.1. Примеры тензоров ранга 0, 1, 2, 3 и 4

Длина формы называется *рангом* (rank) тензора. Например, ранг одномерного тензора *(вектора)* равен 1. У одномерного тензора форма представляет собой массив, содержащий одно число, соответствующее длине этого одномерного тензора. При увеличении ранга на 1 получается двумерный тензор, который можно наглядно изобразить в виде сетки чисел на двумерной плоскости (в качестве примера можно привести изображение в оттенках серого). Форма двумерного тензора состоит из двух чисел, описывающих высоту и ширину этой сетки. При дальнейшем увеличении ранга на 1 получается трехмерный тензор. Как показано в примере на рис. Б.1, трехмерный тензор можно наглядно изобразить в виде трехмерной сетки чисел. Форма трехмерного тензора состоит из трех чисел, соответствующих размерам этой сетки по трем измерениям. Думаем, закономерность уже ясна. Тензоры ранга 4 (четырехмерные тензоры) непосредственно визуализировать сложнее, поскольку мы живем в трехмерном мире. Четырехмерные тензоры применяются во множестве моделей, например в глубоких сверточных сетях. TensorFlow.js поддерживает тензоры вплоть до ранга 6. На практике тензоры ранга 5 используются лишь в некоторых нишевых сценариях (например, при работе с видеоданными), а тензоры ранга 6 — еще реже.

Б.1.1. Скаляры (тензоры ранга 0)

Скаляр — это тензор, форма которого представляет собой пустой массив (`[]`). У него нет осей координат, и он содержит только одно значение. Создать новый скаляр можно с помощью функции `tf.scalar()`. В консоли JavaScript (опять же предполагая, что TensorFlow.js загружен и доступен под псевдонимом `tf`), сделайте следующее:

```
> const myScalar = tf.scalar(2018);[1]
1
> myScalar.print();
Tensor
    2018
> myScalar.dtype;
"float32"
> myScalar.shape;
[]
> myScalar.rank;
0
```

Мы создали скалярный тензор, содержащий одно значение `2021`. Его форма — пустой список, как и следовало ожидать. dtype у него по умолчанию — `"float32"`. Чтобы dtype был целым числом, необходимо указать дополнительный аргумент `'int32'` при вызове функции `tf.scalar()`:

```
> const myIntegerScalar = tf.scalar(2021, 'int32');
> myIntegerScalar.dtype;
"int32"
```

Для извлечения данных обратно из тензора можно воспользоваться асинхронным методом `data()`. Этот метод асинхронный, поскольку в общем случае тензор может храниться не только в основной памяти, а, например, в GPU, в виде текстуры WebGL. Извлечение значений подобных тензоров требует операций, которые могут не выполняться сразу же, а нам не хотелось бы, чтобы они блокировали основной поток выполнения JavaScript. Именно поэтому метод `data()` — асинхронный. Существует также синхронная функция, предназначенная для извлечения значений тензоров с помощью опроса: `dataSync()`. Этот метод удобен, но блокирует основной поток выполнения JavaScript, так что использовать его следует осмотрительно (например, во время отладки). Лучше использовать везде, где можно, асинхронный метод `data()`:

```
> arr = await myScalar.data();
Float32Array [2018]
> arr.length
1
> arr[0]
2018
```

Пример использования `dataSync()`:

```
> arr = myScalar.dataSync();
Float32Array [2018]
> arr.length
1
> arr[0]
2018
```

[1] Обратите внимание, что ради экономии места и большей ясности мы опустили строки, выведенные в консоль JavaScript в результате присваиваний, поскольку они для наших целей неважны.

Как видим, для тензоров типа float32 методы `data()` и `dataSync()` возвращают значения в виде типа `Float32Array` JavaScript. Наверное, это удивит вас, особенно если вы ждали обычное число, но представляется логичным, если учесть, что для тензоров других типов может понадобиться вернуть контейнер, содержащий несколько чисел. Для типа int32 и тензоров булева типа `data()` и `dataSync()` возвращают `Int32Array` и `Uint8Array` соответственно.

Обратите внимание, что, хотя скаляр содержит всегда ровно один элемент, обратное неверно. Тензор ранга больше 0 может также содержать ровно один элемент, если произведение чисел в его форме равно 1. Например, двумерный тензор формы `[1, 1]` содержит один элемент, но обладает двумя осями координат.

Б.1.2. tensor1d (тензоры ранга 1)

Одномерные тензоры иногда называют тензорами ранга 1 или векторами. Одномерный тензор включает ровно одну ось координат, а его форма — массив длиной 1. Следующий код создает вектор в консоли:

```
> const myVector = tf.tensor1d([-1.2, 0, 19, 78]);
> myVector.shape;
[4]
> myVector.rank;
1
> await  myVector.data();
Float32Array(4) [-1.2, 0, 19, 78]
```

Этот одномерный тензор содержит четыре элемента, а потому его можно назвать четырехмерным вектором. Не путайте четырехмерный *вектор* с четырехмерным *тензором*! Четырехмерный вектор — это одномерный тензор с одной осью координат, содержащий ровно четыре значения, в то время как у четырехмерного тензора — четыре оси координат (причем число измерений по каждой оси координат может быть любым). Размерность может означать либо число элементов по конкретной оси координат (как в нашем четырехмерном векторе), либо число осей координат в тензоре (например, четырехмерный тензор). Иногда это может приводить к путанице. Формально правильнее и менее двусмысленно будет говорить о тензорах ранга 4, но двусмысленное название «четырехмерный тензор», тем не менее встречается очень часто. В большинстве случаев это не вызывает проблем, поскольку из контекста обычно понятно, что имеется в виду.

Как и в случае скалярных тензоров, для обращения к значениям элементов одномерного тензора можно использовать методы `data()` и `dataSync()`, например:

```
> await myVector.data()
Float32Array(4) [-1.2000000476837158, 0, 19, 78]
```

Или можно использовать синхронную версию `data()` — вышеупомянутый метод `dataSync()`, — но учтите, что `dataSync()` может блокировать поток выполнения UI и его следует по возможности избегать:

```
> myVector.dataSync()
Float32Array(4) [-1.2000000476837158, 0, 19, 78]
```

Для доступа к значению конкретного элемента одномерного тензора можно просто обратиться к нему по индексу TypedArray, возвращаемому методами `data()` или `dataSync()`, например:

```
> [await myVector.data()][2]
19
```

Б.1.3. tensor2d (тензоры ранга 2)

Двумерный тензор содержит две оси координат. В некоторых случаях двумерные тензоры называют *матрицами*, а его две оси координат интерпретируют как индексы строк и столбцов матрицы соответственно. Матрицу можно наглядно изобразить в виде прямоугольной сетки элементов (см. третий блок на рис. Б.1). В TensorFlow.js

```
> const myMatrix = tf.tensor2d([[1, 2, 3], [40, 50, 60]]);
> myMatrix.shape;
[2, 3]
> myMatrix.rank;
2
```

Записи на первой оси координат называются *строками* (rows), а на второй — *столбцами* (columns). В предыдущем примере первая строка выглядит как `[1, 2, 3]`, а первый столбец — `[1, 40]`. Важно отдавать себе отчет, что методы `data()` и `dataSync()` возвращают данные в виде «плоских» массивов *в построчном порядке* (row-major). Другими словами, сначала в `Float32Array` будут находиться элементы первой строки, за которыми будут следовать элементы второй строки и т. д.[1]:

```
> await myMatrix.data();
Float32Array(6) [1, 2, 3, 40, 50, 60]
```

Ранее мы упоминали, что методы `data()` и `dataSync()` с последующим индексом можно использовать для доступа к значению любого элемента одномерного тензора. Индексация при работе с двумерными тензорами усложняется, поскольку объект TypedArray, возвращаемый методами `data()` или `dataSync()`, схлопывает элементы двумерного тензора. Например, для обращения к элементу TypedArray, соответствующему элементу во второй строке и втором столбце двумерного тензора, необходимо выполнить следующие арифметические операции:

```
> (await myMatrix.data())[1 * 3 + 1];
50
```

К счастью, TensorFlow.js предоставляет дополнительные методы для извлечения значений из тензоров в обычные структуры данных JavaScript: `array()` и `arraySync()`. В отличие от `data()` и `dataSync()` эти методы возвращают вложенные JavaScript-массивы, сохраняющие ранг и формы исходных тензоров. Например:

```
> JSON.stringify(await myMatrix.array())
 "[[1,2,3],[40,50,60]]"
```

[1] В отличие от упорядочения данных по столбцам, встречающегося в некоторых фреймворках численной обработки, например MATLAB и R.

Для обращения к элементу во второй строке и втором столбце достаточно произвести двойную индексацию вложенного массива:

```
> (await myMatrix.array())[1][1]
 50
```

Благодаря этому не требуются арифметические операции с индексами, что особенно удобно для многомерных тензоров. `arraySync()` — синхронная версия `array()`. Как и `dataSync()`, `arraySync()` может блокировать поток выполнения UI и потому использовать ее следует осмотрительно.

В вызове `tf.tensor2d()` мы передали в качестве аргумента вложенный JavaScript-массив. Этот аргумент состоит из строк массивов, вложенных в другой массив. На основе этой вложенной структуры `tf.tensor2d()` выясняет форму двумерного тензора — количество строк и столбцов в нем соответственно. Можно также создать такой же двумерный тензор с помощью `tf.tensor2d()`, передав элементы в виде плоского (невложенного) JavaScript-массива и указав дополнительно второй аргумент, определяющий форму двумерного тензора:

```
> const myMatrix = tf.tensor2d([1, 2, 3, 40, 50, 60], [2, 3]);
> myMatrix.shape;
[2, 3]
> myMatrix.rank;
2
```

При таком подходе произведение всех чисел аргумента `shape` должно равняться числу элементов в массиве чисел, иначе во время вызова `tf.tensor2d()` будет возвращена ошибка. Тензоры более высоких рангов также можно создавать двумя аналогичными способами: либо с помощью одного вложенного массива, либо с помощью плоского массива, сопровождаемого аргументом `shape`. В различных примерах в книге используются оба подхода.

Б.1.4. Тензоры ранга 3 и более высоких рангов

Если упаковать несколько двумерных тензоров в новый массив, получится трехмерный тензор, который можно представить себе в виде куба элементов (четвертый блок на рис. Б.1). Тензоры ранга 3 создаются в TensorFlow.js аналогично предыдущим рангам:

```
> const myRank3Tensor = tf.tensor3d([[[1, 2, 3],
                                      [4, 5, 6]],
                                      [[10, 20, 30],
                                      [40, 50, 60]]]);
> myRank3Tensor.shape;
[2, 2, 3]
> myRank3Tensor.rank;
3
```

Другой способ сделать то же самое — передать плоский (невложенный) массив значений вместе с его формой, указанной явным образом:

```
> const anotherRank3Tensor = tf.tensor3d(
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
    [2, 2, 3]);
```

Функцию `tf.tensor3d` в этом примере можно заменить более общей функцией `tf.tensor()`. С ее помощью можно создавать тензоры вплоть до ранга 6. В следующем примере создаются тензор ранга 3 и тензор ранга 6:

```
> anotherRank3Tensor = tf.tensor(
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12],
    [2, 2, 3]);
> anotherRank3Tensor.shape;
[2, 2, 3]
> anotherRank3Tensor.rank;
3
> tinyRank6Tensor = tf.tensor([13], [1, 1, 1, 1, 1, 1]);
> tinyRank6Tensor.shape;
[1, 1, 1, 1, 1, 1]
> tinyRank6Tensor.rank;
6
```

Б.1.5. Понятие батчей данных

На практике первая ось координат (ось 0, поскольку индексы начинаются с 0) во всех тензорах, которые только могут встретиться вам в глубоком обучении, практически всегда будет *осью батчей* (batch axis, иногда ее называют *осью примеров данных* (samples axis) или *измерением батчей* (batch dimension)). Поэтому ранг подаваемого на вход модели тензора будет на единицу превышать ранг отдельных входных признаков. Это справедливо для всех моделей TensorFlow.js в этой книге. Размер этого первого измерения равен числу примеров данных в батче, называемому *размером батча* (batch size). Например, в примере классификации ирисов из главы 3 (см. листинг 3.9) входной признак каждого из примеров состоит из четырех чисел в виде вектора длиной 4 (одномерный тензор формы `[4]`). Поэтому входной сигнал модели классификации ирисов — двумерный, формы `[null, 4]`, в которой первый `null` указывает, что размер пакета будет выяснен во время работы модели (рис. Б.2). Это соглашение о батчах применимо также к выходному сигналу модели. Например, выходной сигнал модели классификации ирисов представляет собой унитарное представление трех возможных типов ирисов для каждого отдельного входного признака — одномерный тензор формы `[3]`. Однако фактически выходной сигнал модели здесь — двумерный тензор формы `[null, 3]`, в которой первое измерение со значением `null` соответствует размеру пакета, которое будет определено позднее.

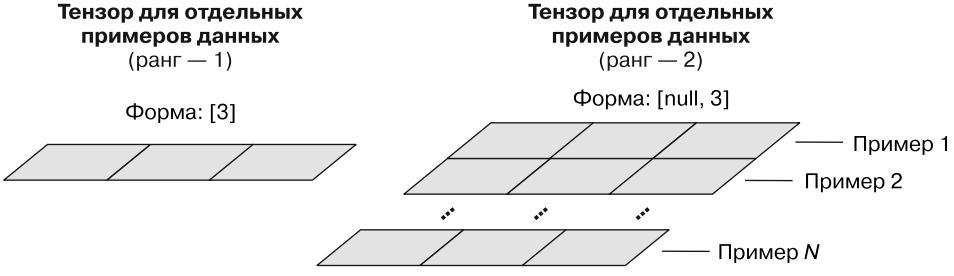


Рис. Б2. Формы тензоров для отдельных примеров данных *(слева)* и батчей примеров данных *(справа)*. Ранг тензора для батчей примеров данных на 1 превышает ранг тензора отдельных примеров данных. Именно такой формат используется в качестве параметров методов predict(), fit() и evaluate() объектов tf.Model. null в форме тензора для примеров данных указывает, что размер первого измерения тензора не определен, так что при реальных вызовах вышеупомянутых методов может быть равен любому положительному целому числу

Б.1.6. Примеры тензоров из практики

Давайте покажем несколько более реалистичных примеров тензоров, подобных тем, которые встретятся вам в этой книге. Данные, с которыми вы будете работать, практически всегда будут относиться к одной из следующих категорий. Выше мы всегда следовали соглашению о наименовании батчей и всегда указывали число примеров данных в качестве первого измерения батча (`числоПримеров`).

- *Векторные данные* — двумерные тензоры формы `[числоПримеров, признаки]`.
- Данные в виде *временных рядов (последовательностей)* — трехмерные тензоры формы `[числоПримеров, интервалы времени, признаки]`.
- *Изображения* — 4-мерные тензоры формы `[числоПримеров, высота, ширина, каналы]`.
- *Видеоданные* — 5-мерные тензоры формы `[числоПримеров, кадр, высота, ширина, каналы]`.

Векторные данные

Наиболее распространенный вид данных. В подобных наборах данных отдельные примеры данных кодируются векторами, а потому батчи данных можно кодировать тензорами ранга 2, в которых роль первой оси координат играет ось примеров данных, а второй — ось признаков.

Рассмотрим два примера.

- Набор страховых данных о людях, в котором учитывается возраст каждого из страхуемых, почтовый индекс и доход. Каждый человек характеризуется вектором из трех значений, так что весь набор данных по 100 000 человек хранится в двумерном тензоре формы `[100000, 3]`.
- Набор данных текстовых документов, где каждому документу соответствует количество вхождений в него каждого слова (например, исходя из словаря 20 000 наиболее распространенных слов русского языка). Каждый документ можно за-

кодировать вектором из 20 000 значений (по одному на каждое слово словаря), а значит, батч из 500 документов можно хранить в тензоре формы `[500, 20000]`.

Временные ряды (последовательности)

Любые данные, в которых должно учитываться время (или в любом случае, когда последовательность упорядочена), имеет смысл хранить в трехмерном тензоре с явной осью координат для времени. Примеры данных кодируются последовательностями векторов (двумерных тензоров), а значит, батч примеров данных можно закодировать трехмерным тензором (рис. Б.3).

Ось времени — почти всегда вторая в тензоре (ось с индексом 1) по общепринятому соглашению, как в следующих примерах.

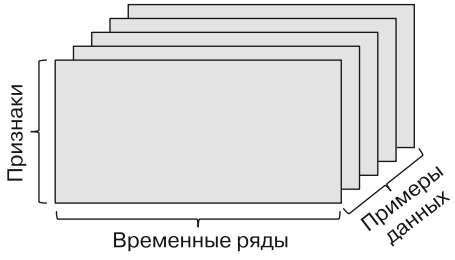


Рис. Б.3. Трехмерный тензор данных для временных рядов

- Набор данных по курсам акций. Каждую минуту сохраняется текущая цена акции, максимальная цена за прошедшую минуту и минимальная цена за прошедшую минуту. Таким образом, данные за каждую минуту кодируются вектором из трех значений. А поскольку в часе 60 минут, то час биржевых торгов кодируется двумерным тензором формы `[60, 3]`. Поэтому форма набора данных по 250 независимым часам будет `[250, 60, 3]`.
- Набор данных твитов, в котором отдельные твиты кодируются последовательностью 280 символов алфавита, состоящего из 128 уникальных символов. При такой схеме отдельный символ можно закодировать бинарным вектором размером 128 (состоящим из нулей, за исключением единицы на позиции, соответствующей символу). Следовательно, каждый твит можно рассматривать как тензор ранга 2 формы `[280, 128]`. А набор данных о 1 000 000 твитов можно хранить в тензоре формы `[1000000, 280, 128]`.

Изображения

Данные об изображении обычно включают три измерения: высоту, ширину и число цветов. И хотя общепринято, что у изображений в оттенках серого обычно только один цветовой канал, тензоры изображений — всегда ранга 3, с одномерным цветовым каналом для изображений в оттенках серого. Батч из 128 изображений в оттенках серого размером 256 × 256, таким образом, можно хранить в тензоре формы `[128, 256, 256, 1]`, а батч из 128 полноцветных изображений — в тензоре формы `[128, 256, 256, 3]` (рис. Б.4). Это называется соглашением NHWC (см. подробности в главе 4).

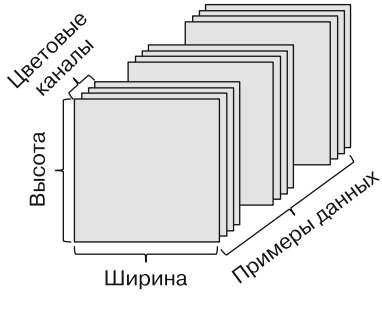


Рис. Б.4. Четырехмерный тензор данных для изображений

В некоторых фреймворках измерение цветовых каналов предшествует высоте и ширине по соглашению NCHW. Мы не станем использовать это соглашение в нашей книге, но не удивляйтесь, если встретите где-то тензор с формой наподобие `[128, 3, 256, 256]`.

Видеоданные

Неформатированные видеоданные — одна из немногих разновидностей встречающихся на практике данных, для которых вам понадобятся тензоры ранга 5. Видеоданные можно рассматривать как последовательность кадров, каждый из которых представляет собой цветное изображение. А поскольку подобный кадр можно хранить в тензоре ранга 3 `[высота, ширина, цветовойКанал]`, то последовательность кадров можно хранить в четырехмерном тензоре формы `[кадры, высота, ширина, цветовойКанал]`, а значит, батч различных изображений можно хранить в пятимерном тензоре формы `[примерыДанных, кадры, высота, ширина, цветовойКанал]`.

Например, 60-секундный видеоролик из YouTube в разрешении 144×256, с дискретизацией четыре кадра в секунду содержит 240 кадров. Батч из четырех таких видеороликов можно хранить в тензоре формы `[4, 240, 144, 256, 3]`. Суммарно 106 168 320 значений! При dtype этого тензора `'float32'` каждое значение занимает 32 бита, так что тензор будет содержать 405 Мбайт данных. Изрядный объем! Встречающиеся на практике видеофайлы обычно занимают намного меньше места, поскольку не хранятся в float32 и обычно достаточно сильно сжаты (например, хранятся в формате MPEG).

Б.1.7. Создание тензоров из тензорных буферов

Мы показали, как создавать тензоры из JavaScript-массивов, с помощью таких функций, как `tf.tensor2d()` и `tf.tensor()`. Для этого необходимо определить значения всех элементов и указать их заранее в JavaScript-массиве. В некоторых случаях, впрочем, создавать подобные JavaScript-массивы с нуля достаточно утомительно. Например, пусть нам нужно создать матрицу 5×5, в которой все внедиагональные элементы — нули, а диагональные элементы формируют нарастающий ряд данных, каждый элемент которого равен индексу строки (столбца) плюс 1:

```
[[1, 0, 0, 0, 0],
 [0, 2, 0, 0, 0],
 [0, 0, 3, 0, 0],
 [0, 0, 0, 4, 0],
 [0, 0, 0, 0, 5]]
```

Код создания вложенного JavaScript-массива, соответствующего этим требованиям, выглядит примерно следующим образом:

```
const n = 5;
const matrixArray = [];
for (let i = 0; i < 5; ++i) {
```

```
  const row = [];
  for (let j = 0; j < 5; ++j) {
    row.push(j === i ? i + 1 : 0);
  }
  matrixArray.push(row);
}
```

После этого наконец можно преобразовать вложенный JavaScript-массив `matrixArray` в двумерный тензор:

```
> const matrix = tf.tensor2d(matrixArray);
```

Весьма длинный код, включающий два вложенных цикла `for`. Можно ли как-то его упростить? Да, безусловно: можно воспользоваться методом `tf.tensorBuffer()` для создания объекта `TensorBuffer`. Элементы объекта `TensorBuffer` можно задавать по индексам и менять их значения с помощью метода `set()`, в отличие от тензорных объектов в TensorFlow.js, значения элементов которых — *неизменяемые* (immutable). По завершении задания значений всех нужных элементов `TensorBuffer` можно легко преобразовать `TensorBuffer` в настоящий тензорный объект с помощью его метода `toTensor()`. Следовательно, новый код, использующий метод `tf.tensorBuffer()` для создания такого же тензора, как и в предыдущем коде, будет иметь вид:

```
const buffer = tf.tensorBuffer([5, 5]);
for (let i = 0; i < 5; ++i) {
  buffer.set(i +  1, i, i);
}
const matrix = buffer.toTensor();
```

Задаем форму тензора при создании объекта TensorBuffer. После создания объект TensorBuffer содержит только нулевые значения

Первый аргумент содержит нужные значения, а остальные аргументы представляют собой индексы задаваемых элементов

Получаем из TensorBuffer сам объект тензора

Таким образом, благодаря методу `tf.tensorBuffer()` число строк снизилось с десяти до пяти.

Б.1.8. Создание тензоров, содержащих одних нули и одни единицы

Очень часто бывает нужно создать тензор заданной формы, все элементы которого равны 0. Для этого предназначена функция `tf.zeros()`. При ее вызове необходимо указать желаемую форму тензора в качестве аргумента:

```
> const x = tf.zeros([2, 3, 3]);
> x.print();
Tensor
    [[[0, 0, 0],
      [0, 0, 0],
      [0, 0, 0]],
      [[0, 0, 0],
      [0, 0, 0],
      [0, 0, 0]]]
```

Тензор был создан с dtype по умолчанию (float32). Для создания содержащих одни нули тензоров с другими dtype необходимо указать dtype в качестве второго аргумента функции `tf.zeros()`.

Кроме того, существует функция `tf.zerosLike()`, с помощью которой можно создать содержащий одни нули тензор с такой же формой и dtype, как и уже существующий. Например, вызов:

```
> const y = tf.zerosLike(x);
```

эквивалентен:

```
> const y = tf.zeros(x.shape, x.dtype);
```

но более лаконичен.

Аналогичные методы существуют для создания тензоров, в которых все элементы равны 1: `tf.ones()` и `tf.onesLike()`.

Б.1.9. Создание тензоров со случайными значениями

Во многих случаях может пригодиться возможность создавать тензоры со случайными значениями, например, для инициализации весовых коэффициентов. Чаще всего для этой цели применяются функции `tf.randomNormal()` и `tf.randomUniform()`. Синтаксис у этих двух функций схожий, различаются только вероятностные распределения значений элементов. Как ясно из названия, `tf.randomNormal()` возвращает тензоры, в которых элементы подчиняются нормальному (гауссовому) распределению[1]. При вызове этой функции только с одним аргументом — формой, получится тензор, элементы которого подчиняются *нормированному* нормальному распределению (unit normal distribution): нормальному распределению с математическим ожиданием = 0 и среднеквадратичным отклонением = 1. Например:

```
> const x = tf.randomNormal([2, 3]);
> x.print():
Tensor
    [[-0.2772508, 0.63506  , 0.3080665],
     [0.7655841 , 2.5264773, 1.142776 ]]
```

При потребности в отличных от используемых по умолчанию математическом ожидании или среднеквадратичном отклонении можно указать их в качестве второго и третьего входных аргументов соответственно. Например, следующий вызов создает тензор, в котором элементы подчиняются нормальному распределению с математическим ожиданием = –20 и среднеквадратичным отклонением = 0,6:

```
> const x = tf.randomNormal([2, 3], -20, 0.6);
> x.print();
Tensor
    [[-19.0392246, -21.2259483, -21.2892818],
     [-20.6935596, -20.3722878, -20.1997948]]
```

[1] Для тех читателей, которые хорошо разбираются в статистике: значения элементов независимы друг от друга.

С помощью функции `tf.randomUniform()` можно создавать случайные тензоры с равномерно распределенными значениями элементов. По умолчанию равномерное распределение — нормированное, то есть его нижняя граница равна 0, а верхняя граница — 1:

```
> const x = tf.randomUniform([3, 3]);
> x.print();
Tensor
    [[0.8303654, 0.3996494, 0.3808384],
     [0.0751046, 0.4425731, 0.2357403],
     [0.4682371, 0.0980235, 0.7004037]]
```

Если же требуется, чтобы элементы подчинялись ненормированному равномерному распределению, можно указать нижнюю и верхнюю границы в виде второго и третьего аргументов функции `tf.randomUniform()` соответственно. Например:

```
> const x = tf.randomUniform([3, 3], -10, 10);
```

создает тензор, значения в котором случайно равномерно распределены по интервалу `[-10, 10)`:

```
> x.print();
Tensor
    [[-7.4774652, -4.3274679, 5.5345411 ],
     [-6.767087 , -3.8834026, -3.2619202],
     [-8.0232048, 7.0986223 , -1.3350322]]
```

Функцию `tf.randomUniform()` можно использовать для создания тензоров типа int32 со случайными значениями. Это удобно при необходимости генерации случайных меток. Например, следующий код создает вектор длиной 10, значения в котором случайным образом выбраны из целых чисел от 0 до 100 (интервал `[0, 100)`):

```
> const x = tf.randomUniform([10], 0, 100, 'int32');
> x.print();
Tensor
    [92, 16, 65, 60, 62, 16, 77, 24, 2, 66]
```

Обратите внимание, что главное в этом примере — аргумент `'int32'`. Без него тип значений в тензоре будет float32, а не int32.

Б.2. Основные операции над тензорами

От тензоров не было бы никакого проку, если бы над ними нельзя было производить различные операции. TensorFlow.js поддерживает множество операций над тензорами. Список их, вместе со всей документацией, можно найти по адресу https://js.tensorflow.org/api/latest. Описывать их все утомительно и излишне. Поэтому мы лишь приведем примеры чаще всего используемых из них. Наиболее используемые операции над тензорами делятся на два типа: унарные и бинарные. *Унарная* операция получает на входе один тензор и возвращает также один тензор, а *бинарная* получает на входе два тензора и возвращает новый тензор.

Б.2.1. Унарные операции

Рассмотрим операцию получения отрицательного тензора для данного, то есть тензора, все значения элементов которого отрицательны по отношению к соответствующим элементам исходного тензора, — и формирования нового тензора такой же формы и с таким же dtype. Сделать это можно с помощью функции `tf.neg()`:

```
> const x = tf.tensor1d([-1, 3, 7]);
> const y = tf.neg(x);
> y.print();
Tensor
    [1, -3, -7]
```

Функциональные API и цепочечные API

В предыдущем примере мы вызвали функцию `tf.neg()` с тензором `x` в качестве аргумента. TensorFlow.js позволяет описывать математически эквивалентную операцию более лаконично, с помощью метода `neg()` самого объекта тензора, вместо функции из пространства имен `tf.*`:

```
> const y = x.neg();
```

В таком простом примере мы, кажется, не так уж много сэкономили кода благодаря этому новому API. Впрочем, в случаях, когда применяется одна операция за другой, второй вариант API демонстрирует значительное превосходство над первым. Например, представьте себе гипотетический алгоритм, в котором необходимо найти отрицательный к `x` тензор, затем вычислить обратный (тензор, в котором каждый элемент равен 1, деленной на соответствующий элемент исходного) к полученному тензору и применить к нему функцию активации ReLU. Для реализации указанного алгоритма с помощью первого API понадобится следующий код:

```
> const y = tf.relu(tf.reciprocal(tf.neg(x)));
```

Напротив, при использовании второго API код выглядит так:

```
> const y = x.neg().reciprocal().relu();
```

Вторая реализация превосходит первую в нескольких аспектах.

- Необходимо набрать меньше символов кода, а потому меньше и вероятность ошибиться в нем.
- Не нужно заботиться о правильном числе вложенных пар открывающихся и закрывающихся скобок (хотя большинство современных редакторов кода значительно упрощает это для разработчика).
- Что важнее, порядок указания методов в коде соответствует порядку соответствующих математических операций (обратите внимание, что в первом варианте порядок операций — обратный). Благодаря этому удобочитаемость кода во втором варианте выше.

Мы будем называть первый API *функциональным* (functional), поскольку в его основе — вызов функций из пространства имен `tf.*`. Второй API мы будем называть

цепочечным (chaining), поскольку операции в нем идут одна за другой, как бы цепочкой (как вы видели в предыдущем примере). Большинство операций в TensorFlow.js доступны как в функциональной версии, в пространстве имен `tf.*`, так и в цепочечной, в виде методов объектов тензоров. Вы можете выбирать любую из этих версий, в зависимости от своих потребностей. В книге мы используем оба API в различных местах, отдавая предпочтение цепочечному API для последовательных операций.

Поэлементные и сверточные операции

Общее свойство всех упомянутых примеров унарных операций (`tf.neg()`, `tf.reciprocal()` и `tf.relu()`) — независимое выполнение операций над отдельными элементами входных тензоров. В результате форма возвращаемого этими операциями тензора совпадает с формой входного тензора. Впрочем, многие унарные операции в TensorFlow.js приводят к уменьшению формы тензора, по сравнению с исходным. Что означает «уменьшение» в контексте форм тензоров? В некоторых случаях — меньший ранг. Например, унарная операция, получающая на входе трехмерный тензор (тензор ранга 3), может возвращать скаляр (тензор ранга 0). В других случаях оно может значить, что размер определенного измерения меньше, чем исходного. Например, унарная операция, получив на входе тензор формы `[3, 20]`, может вернуть тензор формы `[3, 1]`. Вне зависимости от вида сжатия формы, такие операции называются *сверточными* (reduction operations).

Одна из чаще всего используемых операций свертки — `tf.mean()`. В цепочечном API она доступна в качестве метода `mean()` класса `Tensor`. При вызове без каких-либо дополнительных аргументов, она вычисляет арифметическое среднее всех элементов входного тензора, вне зависимости от его формы, и возвращает скаляр. В цепочечном API она используется примерно так:

```
> const x = tf.tensor2d([[0, 10], [20, 30]]);
> x.mean().print();
Tensor
    15
```

Иногда необходимо вычислить среднее значение по отдельным строкам двумерного тензора (матрицы), а не по тензору в целом. Для этого можно передать в метод `mean()` дополнительный аргумент:

```
> x.mean(-1).print();
Tensor
    [5, 25]
```

Аргумент `-1` означает, что метод `mean()` должен вычислять арифметические средние значения по последнему измерению тензора[1]. Это измерение называется в подобном случае *свертываемым измерением* (reduction dimension), поскольку оно сворачивается в выходном тензоре, который оказывается тензором ранга 1. Указать сверточное измерение можно также с помощью его индекса:

```
> x.mean(1).print();
```

[1] Что соответствует принятому в языке Python соглашению об индексах.

Обратите внимание, что метод `mean()` позволяет указывать и несколько сверточных измерений. Например, если нужно вычислить для трехмерного входного тензора формы `[10, 6, 3]` арифметическое среднее по последним двум измерениям и получить в результате тензор формы `[10]`, можно вызвать метод `mean()` в виде `x.mean([-2, -1])` или `x.mean([1, 2])`. Мы оставим это читателю в качестве упражнения в конце этого приложения.

В числе других часто используемых унарных операций свертки:

- метод `tf.sum()` — практически идентичен `tf.mean()`, но вычисляет сумму, а не среднее значение, элементов;
- метод `tf.norm()`, вычисляющий норму элементов. Существует множество видов норм. Например, норма L1 представляет собой сумму модулей элементов. Норма L2 вычисляется путем извлечения квадратного корня из суммы квадратов элементов, другими словами, представляет собой длину вектора в евклидовом пространстве. С помощью метода `tf.norm()` можно вычислять дисперсию или среднеквадратичное отклонение списка чисел;
- методы `tf.min()` и `tf.max()`, вычисляющие минимальное и максимальное значения элементов соответственно;
- метод `tf.argMax()`, возвращающий индекс максимального элемента по оси свертки. Эту операцию часто применяют для преобразования вероятностей классов в индекс наилучшего класса (например, см. задачу классификации ирисов в разделе 3.3.2). Аналогичную функциональность поиска минимального значения предоставляет метод `tf.argMin()`.

Мы уже упоминали, что поэлементные операции сохраняют форму входного тензора. Но обратное утверждение не соответствует действительности. Некоторые сохраняющие форму входного тензора операции — вовсе не поэлементные. Например, операция `tf.transpose()` производит транспонирование матрицы, при котором элемент с индексами `[i, j]` во входном двумерном тензоре переводится в элемент с индексами `[j, i]` в выходном двумерном тензоре. Входная и выходная формы при использовании метода `tf.transpose()` идентичны, если входной тензор представляет собой квадратную матрицу, но это вовсе не поэлементная операция, так как значение элемента `[i, j]` выходного тензора зависит не только от значения элемента `[i, j]` входного тензора, а от значений на других позициях.

Б.2.2. Бинарные операции

В отличие от унарных операций для бинарной операции необходимо два аргумента. Вероятно, чаще всего используемая бинарная операция `tf.add()`. Она же, вероятно, и самая простая, так как просто складывает два тензора. Например:

```
> const x = tf.tensor2d([[0, 2], [4, 6]]);
> const y = tf.tensor2d([[10, 20], [30, 46]]);
> tf.add(x, y).print();
Tensor
    [[10, 22],
     [34, 52]]
```

Некоторые аналогичные бинарные операции:

- метод `tf.sub()` для вычитания одного тензора из другого;
- метод `tf.mul()` для умножения двух тензоров;
- метод `tf.matMul()` для вычисления матричного произведения двух тензоров;
- методы `tf.logicalAnd()`, `tf.logicalOr()` и `tf.logicaXor()` для выполнения соответственно логических операций И, ИЛИ и исключающее ИЛИ над тензорами булева типа.

Некоторые бинарные операции поддерживают *транслирование* (broadcasting), то есть работу с двумя входными тензорами различной формы с применением элемента входного тензора меньшей формы к нескольким элементам второго входного тензора в соответствии с определенным правилом. См. подробное обсуждение в инфобоксе 2.4.

Б.2.3 Конкатенация и срезы тензоров

Унарные и бинарные операции относятся к типу «тензор на входе, тензор на выходе» (tensor-in-tensor-out, TITO) в том смысле, что получают один или несколько тензоров в качестве входных данных и возвращают тензор в качестве выходных данных. Некоторые часто используемые операции в TensorFlow.js не относятся к этому типу, поскольку получают на входе тензор, наряду с другим (-и) нетензорным (-и) аргументом (-ами) в качестве входных данных. Вероятно, чаще всего используемая функция из этой категории — `tf.concat()`. С ее помощью можно «склеивать» несколько тензоров совместимой формы в один. Например, можно склеить тензор формы `[5, 3]` с тензором формы `[4, 3]` по первой оси и получить тензор `[9, 3]`, но невозможно объединить тензоры форм `[5, 3]` и `[4, 2]`! Функцию `tf.concat()` можно использовать для конкатенации тензоров при условии совместимости форм. Например, следующий код склеивает состоящий из одних нулей тензор формы `[2, 2]` с состоящим из одних единиц тензором формы `[2, 2]`, в результате чего получается тензор формы `[4, 2]`, в котором «верхняя» половина заполнена нулями, а «нижняя» — единицами:

```
> const x = tf.zeros([2, 2]);
> const y = tf.ones([2, 2]);
> tf.concat([x, y]).print();
Tensor
    [[0, 0],
     [0, 0],
     [1, 1],
     [1, 1]]
```

Поскольку формы обоих входных тензоров идентичны, можно склеить их иначе: по второй оси. Ось можно указать с помощью второго входного аргумента метода `tf.concat()`. В результате получится тензор формы `[2, 4]`, в котором левая половина заполнена нулями, а правая — единицами:

```
> tf.concat([x, y], 1).print();
Tensor
    [[0, 0, 1, 1],
     [0, 0, 1, 1]]
```

Помимо конкатенации нескольких тензоров в один, иногда бывает нужно произвести и «обратную» операцию — извлечь часть тензора. Например, пусть мы создали двумерный тензор (матрицу) формы `[3, 2]`:

```
> const x = tf.randomNormal([3, 2]);
> x.print();
Tensor
    [[1.2366893 , 0.6011682 ],
     [-1.0172369, -0.5025602],
     [-0.6265425, -0.0009868]]
```

и хотим извлечь ее вторую строку. Для этого можно воспользоваться цепочечной версией метода `tf.slice()`:

```
> x.slice([1, 0], [1, 2]).print();
Tensor
     [[-1.0172369, -0.5025602],]
```

Первый аргумент метода `tf.slice()` указывает, что нужная нам часть входного тензора начинается с индекса 1 по первому измерению и индекса 0 по второму измерению. Другими словами, она должна начинаться со второй строки и первого столбца, поскольку наш тензор представляет собой матрицу. Второй аргумент задает форму желаемых выходных данных: `[1, 2]`, или, на языке матриц, одна строка и два столбца.

Как вы можете проверить по выводимым значениям, мы успешно извлекли вторую строку матрицы 3×2. Ранг выходных данных — такой же, как и у входных (2), но размер первого измерения — 1. В данном случае мы полностью извлекли все второе измерение (все столбцы) и подмножество первого измерения (подмножество строк). Это особый случай, позволяющий достичь того же результата с помощью более простого синтаксиса:

```
> x.slice(1, 1).print();
Tensor
     [[-1.0172369, -0.5025602],]
```

При таком более простом синтаксисе необходимо только указать начальный индекс и размер нужного фрагмента по первому измерению. Если передать 2, а не 1 в качестве второго входного аргумента, выходные данные будут включать вторую и третью строки матрицы:

```
> x.slice(1, 2).print();
Tensor
    [[-1.0172369, -0.5025602],
     [-0.6265425, -0.0009868]]
```

Как вы можете предположить, этот более простой синтаксис связан с соглашением о батчах. Он упрощает извлечение данных для отдельных примеров данных из тензора батча.

Но что, если нам нужно извлечь *столбцы* матрицы, а не строки? В этом случае нам пришлось бы использовать более сложный синтаксис. Например, пусть нам нужен второй столбец матрицы. Это можно сделать с помощью:

```
> x.slice([0, 1], [-1, 1]).print();
Tensor
    [[0.6011682 ],
     [-0.5025602],
     [-0.0009868]]
```

Здесь первый аргумент (`[0, 1]`) представляет собой массив, соответствующий начальным индексам нужного среза. Это первый индекс по первому измерению и второй индекс по второму. Проще говоря, мы хотим, чтобы срез начинался с первой строки и второго столбца. Второй аргумент (`[-1, 1]`) задает размер нужного нам среза. Первое число указывает, что мы хотим получить данные для всех индексов по первому измерению (все строки), а второе число (1) означает, что мы хотим получить данные только для одного индекса по второму измерению (только один столбец). Результат представляет собой второй столбец матрицы.

Из синтаксиса метода `slice()` вы, наверное, поняли, что он подходит для извлечения не только строк и столбцов. На самом деле он достаточно гибок для извлечения произвольной «подматрицы» входного двумерного тензора (любой последовательной прямоугольной области в матрице), если начальные индексы и размер массива указаны должным образом. В общем случае для тензоров произвольного ранга > 0 метод `slice()` позволяет извлечь произвольный непрерывный подтензор того же ранга из входного тензора. Мы оставим это в качестве упражнения для читателей.

Помимо методов `tf.slice()` и `tf.concat()`, существуют еще две часто используемые операции для разбиения тензора на части или объединения нескольких тензоров в один: `tf.unstack()` и `tf.stack()`. Метод `tf.unstack()` разбивает тензор на несколько частей по первому измерению, каждая — размером 1 (по первому измерению). Например, можно воспользоваться цепочечным API `tf.unstack()`, вот так:

```
> const x = tf.tensor2d([[1, 2], [3, 4], [5, 6]]);
> x.print();
Tensor
    [[1, 2],
     [3, 4],
     [5, 6]]
> const pieces = x.unstack();
> console.log(pieces.length);
  3
> pieces[0].print();
Tensor
    [1, 2]
> pieces[1].print();
Tensor
    [3, 4]
> pieces[2].print();
Tensor
    [5, 6]
```

Как вы могли заметить, ранг возвращаемых методом `unstack()` частей на единицу меньше, чем у входного тензора.

Метод `tf.stack()` — противоположность метода `tf.unstack()`. Как ясно из его названия, он «складирует» несколько тензоров одинаковой формы в новый тензор. Продолжая предыдущий фрагмент кода, мы можем вот так совместить части тензора обратно:

```
> tf.stack(pieces).print();
Tensor
    [[1, 2],
     [3, 4],
     [5, 6]]
```

Метод `tf.unstack()` удобен для извлечения из тензора батча данных, соответствующих отдельным примерам данных; метод `tf.stack()` удобен для объединения отдельных примеров данных в тензор батча.

Б.3. Управление памятью в TensorFlow.js: tf.dispose() и tf.tidy()

При работе непосредственно с тензорными объектами в TensorFlow.js необходимо брать на себя управление выделяемой для них памятью. В частности, тензоры необходимо удалять после создания и использования, иначе они продолжат занимать выделенную для них память. В случае слишком большого числа неудаленных тензоров или слишком большого их суммарного размера в конце концов закончится либо память WebGL, доступная вкладке браузера, либо системная память/память GPU, доступная процессу Node.js (в зависимости от того, какая версия tfjs-node используется: CPU или GPU). TensorFlow.js не производит автоматической сборки мусора для создаваемых пользователями тензоров[1]. Причина в том, что JavaScript не поддерживает финализации объектов. TensorFlow.js предоставляет две функции для управления памятью: `tf.dispose()` и `tf.tidy()`.

Например, рассмотрим сценарий с повторяемым многократно с помощью цикла `for` выводом на основе модели TensorFlow.js:

```
const model = await tf.loadLayersModel(
    'https://storage.googleapis.com/tfjs-models/tfjs/iris_v1/model.json');
const x = tf.randomUniform([1, 4]);
for (let i = 0; i < 3; ++i) {
  const y = model.predict(x);
  y.print();
  console.log(`# of tensors: ${tf.memory().numTensors}` );
}
```

Загружаем заранее обученную модель из Интернета

Создаем фиктивный входной тензор

Проверяем число тензоров, под которые в данный момент выделена память

[1] Впрочем, управление памятью тензоров, создаваемых внутри функций и методов объектов TensorFlow.js, осуществляет сама библиотека, так что не нужно заботиться об обертывании вызовов подобных функций/методов в tf.tidy(). Примеры подобных функций: tf.confusionMatrix(), tf.Model.predict() и tf.Model.fit().

Результаты выполнения этого кода выглядят следующим образом:

```
Tensor
     [[0.4286409, 0.4692867, 0.1020722],]
# of tensors: 14
Tensor
     [[0.4286409, 0.4692867, 0.1020722],]
# of tensors: 15
Tensor
     [[0.4286409, 0.4692867, 0.1020722],]
# of tensors: 16
```

Как вы можете видеть из выведенного в консоль, `model.predict()` создает при каждом вызове дополнительный тензор, не удаляемый после завершения итерации. Если этот цикл `for` будет выполняться в течение достаточно большого числа итераций, то в конце концов приведет в ошибке нехватки памяти. А все потому, что выходной тензор `y` не удаляется должным образом, что приводит к утечке памяти. Исправить ситуацию можно двумя способами.

Первый вариант — вызывать метод `tf.dispose()` выходного тензора, когда он больше не нужен:

```
for (let i = 0; i < 3; ++i) {
  const y = model.predict(x);
  y.print();
  tf.dispose(y);
  console.log(`# of tensors: ${tf.memory().numTensors}` );
}
```

Удаляем выходной тензор после использования

Второй вариант — обернуть тело цикла `for` в вызов `tf.tidy()`:

```
for (let i = 0; i < 3; ++i) {
  tf.tidy(() => {
    const y = model.predict(x);
    y.print();
    console.log(`# of tensors: ${tf.memory().numTensors}` );
  });
}
```

Метод tf.tidy() автоматически удаляет все тензоры, созданные внутри передаваемой в него функции, за исключением тензоров, которые эта функция возвращает

При любом из этих подходов вы увидите, что число выделенных тензоров не меняется по мере итераций, а значит, утечки памяти нигде нет. Какой подход предпочтительнее? В принципе, лучше использовать метод `tf.tidy()` (второй подход), поскольку при этом не нужно отслеживать требующие удаления тензоры. `tf.tidy()` — «умная» функция, удаляющая все тензоры, созданные внутри передаваемой в него анонимной функции (за исключением тензоров, возвращаемых этой функцией, — подробнее об этом далее), даже тензоры, не связанные ни с какими объектами JavaScript. Например, слегка изменим предыдущий код вывода, чтобы получать индекс наилучшего класса с помощью `argMax()`:

```
const model = await tf.loadLayersModel(
    'https://storage.googleapis.com/tfjs-models/tfjs/iris_v1/model.json');
const x = tf.randomUniform([1, 4]);
for (let i = 0; i < 3; ++i) {
```

```
  const winningIndex =
      model.predict(x).argMax().dataSync()[0];
  console.log(`winning index: ${winningIndex}`);
  console.log(`# of tensors: ${tf.memory().numTensors}` );
}
```

При работе этого кода вы увидите, что происходит утечка памяти, выделенной не для одного тензора, а для двух:

```
winning index: 0
# of tensors: 15
winning index: 0
# of tensors: 17
winning index: 0
# of tensors: 19
```

Почему происходит утечка памяти, выделенной для двух тензоров на каждой итерации? Дело в том, что строка:

```
   const winningIndex =
       model.predict(x).argMax().dataSync()[0];
```

приводит к созданию двух новых тензоров. Первый из них — результат работы метода `model.predict()`, а второй — возвращаемое `argMax()` значение. Ни один из них не связан ни с какими объектами JavaScript. Они используются сразу же после создания. Эти два тензора «теряются» в том смысле, что не существует JavaScript-объектов, с помощью которых можно было бы на них сослаться. А потому нельзя воспользоваться для их удаления методом `tf.dispose()`. А вот с помощью `tf.tidy()` по-прежнему можно устранить эту утечку памяти, поскольку она ведет учет новых тензоров, вне зависимости от того, связаны ли они с какими-то объектами JavaScript:

```
const model = await tf.loadLayersModel(
    'https://storage.googleapis.com/tfjs-models/tfjs/iris_v1/model.json');
const x = tf.randomUniform([1, 4]);
for (let i = 0; i < 3; ++i) {
  tf.tidy(() => {
    const winningIndex = model.predict(x).argMax().dataSync()[0];
    console.log(`winning index: ${winningIndex}`);
    console.log(`# of tensors: ${tf.memory().numTensors}` );
  });
}
```

tf.tidy() — автоматически удаляет все тензоры, созданные внутри тела передаваемой в него в качестве аргумента анонимной функции, даже если они не связаны ни с какими объектами JavaScript

В этом примере использования `tf.tidy()` функция не возвращает никаких тензоров. Если же функция возвращает тензоры, удалять их нежелательно, ведь они понадобятся позднее. Эта ситуация часто встречается при написании пользовательских операций с тензорами на основе базовых операций TensorFlow.js. Например, пусть нам нужно написать функцию вычисления нормализованного значения входного тензора — то есть тензора с вычтенным средним значением и приведенным к 1 среднеквадратичным отклонением:

```
function normalize(x) {
  const mean = x.mean();
```

```
  const sd = x.norm(2);
  return x.sub(mean).div(sd);
}
```

В чем основная проблема этой реализации?[1] С точки зрения управления памятью здесь происходит утечка памяти, выделенной для трех тензоров: 1) тензора среднего значения; 2) тензора среднеквадратичного отклонения; 3) (более тонкий нюанс) тензора возвращаемого значения вызова `sub()`. Для устранения этой утечки памяти обернем тело функции в `tf.tidy()`:

```
function normalize(x) {
  return tf.tidy(() => {
    const mean = x.mean();
    const sd = x.norm(2);
    return x.sub(mean).div(sd);
  });
}
```

В этом коде `tf.tidy()` производит три действия.

- Автоматически удаляет тензоры, созданные в анонимной функции, но не возвращаемые ею, включая все три упомянутых источника утечек. Мы видели это в предыдущих примерах.
- Определяет, что анонимная функция возвращает результат вызова `div()`, а потому переносит его в свое собственное возвращаемое значение.
- Вместе с тем удалять этот конкретный тензор она не станет, так что его можно будет использовать снаружи вызова `tf.tidy()`.

Как можно видеть, `tf.tidy()` — «умная» и обладающая большими возможностями функция управления памятью. Она широко используется в базе кода самого TensorFlow.js. Также вы не раз встретитесь с ней в примерах в этой книге. Впрочем, у нее есть одно важное ограничение: передаваемая в `tf.tidy()` анонимная функция *не* может быть асинхронной. Для требующего управления памятью асинхронного кода вам придется воспользоваться `tf.dispose()` и самостоятельно отслеживать тензоры, требующие удаления. В подобных случаях можно воспользоваться `tf.memory().numTensor` для проверки количества тензоров, вызывающих утечки. Рекомендуется также писать модульные тесты для контроля отсутствия утечек памяти.

Б.4. Вычисление градиентов

Этот раздел предназначен для тех читателей, кого интересуют возможности вычисления производных и градиентов в TensorFlow.js. В большинстве моделей глубокого обучения в этой книге вычисление производных и градиентов производится «под

[1] У этой реализации есть и другие проблемы. Например, в ней не производится проверка допустимости входного тензора: необходимо проверить, что он содержит хотя бы два элемента, иначе среднеквадратичное отклонение будет равно нулю, что приведет к делению на ноль и к ошибке. Но эти проблемы с обсуждаемым здесь вопросом напрямую не связаны.

капотом» `model.fit()` и `model.fitDataset()`, так что заботиться об этом не надо. Впрочем, в определенных задачах, например поиска максимально активирующих нейроны изображений для сверточных фильтров в главе 7 и обучения с подкреплением в главе 11, приходится вычислять производные и градиенты явным образом. TensorFlow.js предоставляет API для подобных сценариев использования. Начнем с простейшего сценария — функции, принимающей на входе один тензор и возвращающей на выходе также один тензор:

```
const f = x => tf.atan(x);
```

Для вычисления производной функции (`f`) по входной переменной (`x`) мы воспользуемся функцией `tf.grad()`:

```
const df = tf.grad(f);
```

Обратите внимание, что функция `tf.grad()` не возвращает непосредственно значение производной, а возвращает *функцию*, которая является производной от исходной функции (`f`). А уже эту функцию (`df`) можно вызвать с конкретным значением `x` и получить значение `df/dx`. Например, результат выполнения:

```
const x = tf.tensor([-4, -2, 0, 2, 4]);
df(x).print();
```

дает нам правильное значение производной функции `atan()` в точках –4, –2, 0, 2, и 4 по `x` (рис. Б.5):

```
Tensor
    [0.0588235, 0.2, 1, 0.2, 0.0588235]
```

`tf.grad()` можно использовать только для функций с одним тензором на входе. А что делать с функцией с несколькими входными тензорами? Рассмотрим пример функции `h(x, y)`, равной просто произведению двух тензоров:

```
const h = (x, y) => x.mul(y);
```

Сгенерировать функцию, возвращающую частные производные входной функции по всем аргументам, можно с помощью метода `tf.grads()` (с s в конце):

```
const dh = tf.grads(h);
const dhValues = dh([tf.tensor1d([1, 2]), tf.tensor1d([-1, -2])]);
dhValues[0].print();
dhValues[1].print();
```

который возвращает следующие результаты:

```
Tensor
    [-1, -2]
Tensor
    [1, 2]
```

Эти результаты правильны, поскольку частная производная $x * y$ по x равна y, а частная производная $x * y$ по y равна x.

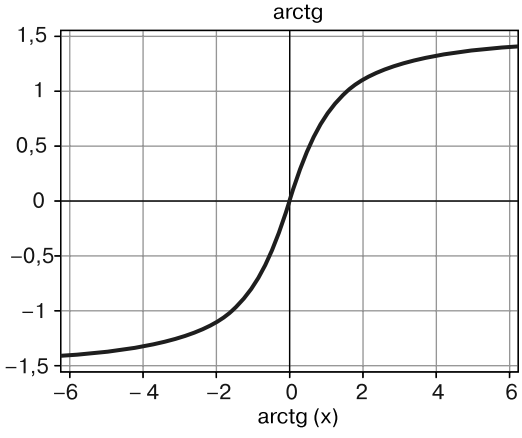


Рис. Б.5. График функции atan(x)

Генерируемые `tf.grad()` и `tf.grads()` функции возвращают только производные, а не возвращаемое значение исходной функции. Например, что, если для *h(x, y)* нам нужно узнать не только производные, но и значение *h*? Для этой цели можно воспользоваться функцией `tf.valueAndGrads()`:

```
const vdh = tf.valueAndGrads(h);
const out = vdh([tf.tensor1d([1, 2]), tf.tensor1d([-1, -2])]);
```

Результат ее (`out`) — объект с двумя полями: `value`, которое содержит значение *h* при заданных входных значениях, и `grads` в том же формате, что и возвращаемое значение сгенерированной `tf.grads()` функции, а именно в виде массива тензоров частных производных:

```
out.value.print();
out.grads[0].print();
out.grads[1].print();
Tensor
    [-1, -4]
Tensor
    [-1, -2]
Tensor
    [1, 2]
```

Все обсуждавшиеся здесь API служат для вычисления производных функций по их явным аргументам. Однако в одном из распространенных сценариев глубокого обучения участвуют функции, использующие в своих вычислениях весовые коэффициенты. Эти весовые коэффициенты представлены объектами `tf.Variable` и *не* передаются явным образом в функции в виде аргументов. И нам нередко приходится вычислять частные производные подобных функций по весам во время обучения. Этот технологический процесс реализуется функцией `tf.variableGrads()`, отслеживающей, к каким обучаемым переменным обращается дифференцируемая

функция, и автоматически вычисляющей производные по ним. Рассмотрим следующий пример:

```
const trainable = true;
const a = tf.variable(tf.tensor1d([3, 4]), trainable, 'a');
const b = tf.variable(tf.tensor1d([5, 6]), trainable, 'b');
const x = tf.tensor1d([1, 2]);
const f = () => a.mul(x.square()).add(b.mul(x)).sum();
const {value, grads} = tf.variableGrads(f);
```

f(a, b) = a * x ^ 2 + b * x. Метод sum() нужен потому, что для tf.variableGrads() необходимо, чтобы дифференцируемая функция возвращала скаляр

Поле `value` возвращаемого `tf.variableGrads()` результата представляет собой значение, возвращаемое `f` при заданных значениях `a`, `b` и `x`. Поле `grads` представляет собой JavaScript-объект, содержащий производные по двум переменным (`a` и `b`) под соответствующими ключевыми именами. Например, производная `f(a, b)` по `a` равна `x ^ 2`, а производная `f(a, b)` по `b` равна `x`:

```
grads.a.print();
grads.b.print();
```

в результате чего получаем правильные значения:

```
Tensor
    [1, 4]
Tensor
    [1, 2]
```

Упражнения

1. Создайте с помощью `tf.tensorBuffer()` единичный четырехмерный тензор, удовлетворяющий следующим свойствам: форма `[5, 5, 5, 5]`, нули на всех позициях, за исключением элементов, индексы которых представляют собой четыре одинаковых числа (например, `[2, 2, 2, 2]`), значения которых должны быть равны 1.
2. Создайте трехмерный тензор формы `[2, 4, 5]` с помощью функции `tf.randomUniform()` с интервалом по умолчанию `[0, 1)`. Напишите, используя `tf.sum()`, строку кода для выполнения свертки-суммирования по второму и третьему измерениям. Посмотрите на результат. Его форма должна быть `[2]`. Как вы думаете, какими примерно должны быть значения элементов? Соответствует ли результат вашим ожиданиям?

   (Подсказка: каково математическое ожидание числа, равномерно распределенного по интервалу `[0, 1)`? Каково математическое ожидание суммы двух таких значений, если они статистически независимы друг от друга?)
3. Создайте матрицу 4×4 с помощью `tf.randomUniform()` (двумерный тензор формы `[4, 4]`). Извлеките из нее с помощью `tf.slice()` подматрицу 2×2, расположенную в центре.

4. Создайте с помощью `tf.ones()`, `tf.mul()` и `tf.concat()` следующий трехмерный тензор: форма `[5, 4, 3]`; значения элементов первого среза по первой оси (тензор формы `[1, 4, 3]`) должны быть все равны 1; значения элементов второго среза по второй оси должны быть все равны 2 и т. д.

   Бонусные баллы: тензор содержит много элементов, так что проверить правильность просто по выведенному функцией `print()` в консоль непросто. Подумайте, как бы вы написали модульный тест для проверки его правильности? (Подсказка: воспользуйтесь методами `data()`, `dataSync()` или `arraySync()`).

5. Напишите JavaScript-функцию, выполняющую следующие операции над двумя двумерными входными тензорами (матрицами) одинаковой формы. Во-первых, сложите эти две матрицы. Во-вторых, разделите полученную матрицу на 2 поэлементно. В-третьих, транспонируйте получившуюся в итоге матрицу. Функция должна возвращать результат транспонирования.

   А. Какими функциями TensorFlow.js вы воспользуетесь для написания этой функции?

   Б. Сможете ли вы реализовать эту функцию дважды: один раз с помощью функционального API, а второй раз — с помощью цепочечного API? Какая реализация выглядит аккуратнее и более удобочитаема?

   В. Что включает в себя транслирование?

   Г. Как предотвратить возможные утечки памяти в этой функции?

   Д. Попробуйте написать модульный тест (с помощью библиотеки Jasmine из https://jasmine.github.io/) для контроля отсутствия утечек памяти.

Франсуа Шолле, Эрик Нильсон, Стэн Байлесчи, Шэнкуинг Цэй

JavaScript для глубокого обучения: TensorFlow.js

Перевел с английского И. Пальти

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*С. Гавриленков*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*Г. Блинов*

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: 06.2021. 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 27.05.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 46,440. Тираж 500. Заказ 0000.