С. Николенко, А. Кадурин, Е. Архангельская

ГЛУБОКОЕ ОБУЧЕНИЕ

ПОГРУЖЕНИЕ В МИР НЕЙРОННЫХ СЕТЕЙ

ПИТЕР®

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

2018

ББК 32.973.236
УДК 004.8
Н63

Николенко С., Кадурин А., Архангельская Е.

Н63 Глубокое обучение. — СПб.: Питер, 2018. — 480 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-02536-2

Перед вами — первая книга о глубоком обучении, написанная на русском языке. Глубокие модели оказались ключом, который подходит ко всем замкам сразу: новые архитектуры и алгоритмы обучения, а также увеличившиеся вычислительные мощности и появившиеся огромные наборы данных привели к революционным прорывам в компьютерном зрении, распознавании речи, обработке естественного языка и многих других типично «человеческих» задачах машинного обучения. Эти захватывающие идеи, вся история и основные компоненты революции глубокого обучения, а также самые современные достижения этой области доступно и интересно изложены в книге. Максимум объяснений, минимум кода, серьезный материал о машинном обучении и увлекательное изложение — в этой уникальной работе замечательных российских ученых и интеллектуалов.

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

ББК 32.973.236
УДК 004.8

ISBN 978-5-496-02536-2

Содержание

Часть I

Как обучать нейронные сети

Глава 1
От биологии к информатике,
или We need to go deeper

TL;DR

Первая глава — вводная. В ней мы:

- узнаем, что в машинном обучении недавно произошла революция и революция эта все еще длится;
- вспомним историю искусственного интеллекта как науки;
- познакомимся с разными задачами машинного обучения, узнаем, как они связаны и чем отличаются друг от друга;
- выясним, почему человеческий мозг может служить образцом для подражания при создании искусственного интеллекта;
- придем к выводу, что нейробиология пока не слишком точная наука;
- закончим кратким обзором самых ярких примеров применения современных нейронных сетей и областей, где им еще есть чему поучиться.

1.1. Революция обучения глубоких сетей

— А как повернулось дело? — задумчиво пробормотал он. — Трах-та-бабах! Революция!

Ему стало весело. Он глотал пахнувший росой и яблоками воздух и думал: «Столб, хлеб, дом, рожь, больница, базар — слова все знакомые, а то вдруг — Революция! Бейте, барабаны!»

А. Гайдар. Бумбараш

Прежде всего, спросим себя, положив руку на сердце:

— Да есть ли у нас сейчас революция?..

Разве та гниль, глупость, дрянь, копоть и мрак, что происходит сейчас, — разве это революция? Революция — сверкающая прекрасная молния, революция — божественно красивое лицо, озаренное гневом Рока, революция — ослепительно яркая ракета, взлетевшая радугой среди сырого мрака!.. Похоже на эти сверкающие образы то, что сейчас происходит?..

А. Аверченко. Дюжина ножей в спину революции

Десять лет назад, в середине 2000-х годов, в машинном обучении началась революция. В 2005–2006 годах группы исследователей под руководством Джеффри Хинтона (Geoffrey Hinton) в университете Торонто и Йошуа Бенджи (Yoshua Bengio) в университете Монреаля научились обучать *глубокие нейронные сети*. И это перевернуло весь мир машинного обучения! Теперь в самых разнообразных предметных областях лучшие результаты получаются с помощью глубоких нейронных сетей. Одним из первых громких промышленных успехов стало распознавание речи: разработанные группой Хинтона глубокие сети очень быстро радикально улучшили результаты распознавания по сравнению с оттачивавшимися десятилетиями классическими подходами, и сегодня любой распознаватель, включая голосовые помощники вроде Apple Siri и Google Now, работает исключительно на глубоких нейронных сетях. А сейчас, к 2017 году, люди научились обучать самые разные архитектуры глубоких нейронных сетей, и те решают абсолютно разные задачи: от распознавания лиц до вождения автомобилей и игры в го.

Но идеи большинства таких моделей появились еще в 80–90-х годах XX века, а то и раньше. Искусственные нейронные сети стали предметом исследований очень давно; скоро мы увидим, что они были одной из первых хорошо оформленных идей искусственного интеллекта, когда и слов-то таких — «искусственный интеллект» — еще никто не слышал. Но с начала 90-х годов XX века до середины нулевых этого[1] нейронные сети были, мягко говоря, не в моде. Известный

[1] Мы искренне надеемся, что написали хорошую книгу, но все-таки не будем здесь делать специальных оговорок в расчете на читателей XXII века...

исследователь нейронных сетей Джон Денкер (John Denker) в 1994 году сказал: «Нейронные сети — это второй лучший способ сделать практически что угодно». И действительно, на тот момент уже было известно, что нейронная сеть теоретически может приблизить любую функцию и обучиться решать любую задачу, а глубокая нейронная сеть способна еще и более эффективно решить гораздо больше разных задач... Но обучать глубокие сети никому не удавалось, другие методы на конкретных практических примерах работали лучше. Из общих методов машинного обучения это были сначала «ядерные методы» (kernel methods), в частности метод опорных векторов, а затем байесовские сети и в целом графические вероятностные модели. Но вообще Денкер вел речь о том, что более простая вероятностная модель, разработанная специально с учетом специфики конкретной задачи, обычно работала лучше, чем нейронная сеть «общего назначения», даже если выбрать ей подходящую для задачи архитектуру. А глубокие нейронные сети обучаться никак не хотели; о том, почему именно, мы подробно поговорим в разделе 3.5.

Решение, предложенное группой Хинтона в середине 2000-х годов, пришло в виде *предобучения без учителя*, когда сеть сначала обучается на большом наборе данных без разметки, а потом уже дообучается на размеченных данных, используя это приближение. Например, если мы хотим распознавать человеческие лица, то давайте сначала пообучаем нейронную сеть на фотографиях с людьми вообще, без разметки (таких фотографий можно легко набрать сколь угодно много), а уже потом, когда сеть «насмотрится» на неразмеченные фотографии, дообучим ее на имеющемся размеченном наборе данных. Оказалось, что при этом конечный результат становится намного лучше, а хорошие и интересные признаки сеть начинает выделять еще на этапе предобучения без учителя. Именно таким образом произошел, в частности, прорыв в распознавании речи. Конечно, здесь пока совершенно непонятно, что сеть собственно должна делать с этими неразмеченными фотографиями, и в этом и заключался прорыв середины нулевых годов. Мы немного поговорим об этих методах в разделе 4.2, однако без подробностей, потому что сегодня эти методы уже практически не используются.

Почему? Потому что все неплохо работает и без них! Оказалось, что сейчас мы можем успешно обучать нейронные сети, в том числе глубокие, фактически теми же методами, которыми раньше это сделать никак не удавалось. Методы предобучения без учителя оказались лишь «спусковым крючком» для революции глубокого обучения. Второй важнейшей причиной стал, собственно, прогресс в вычислительной технике и в размерах доступных для обучения наборов данных. С развитием Интернета данных становилось все больше: например, классический набор данных MNIST, о котором мы начнем подробный разговор в разделе 3.6 и на котором добрый десяток лет тестировались модели компьютерного зрения, — это 70 тысяч изображений рукописных цифр размером 28×28 пикселов, суммарно около 10 Мбайт данных; а современный стандартный датасет для моделей компьютерного зрения ImageNet содержит уже около 1,2 Тбайт изображений. В части

вычислений помогают и закон Мура[1], и важные новшества, которые позволяют обучать нейронные сети быстрее. Вся глава 4 будет посвящена таким новшествам. В основном мы, конечно, будем говорить о продвижениях в самих алгоритмах обучения, но будут и фактически чисто «железные» продвижения: обучение современных нейронных сетей обычно происходит на графических процессорах (GPU, то есть по сути на видеокартах), что часто позволяет ускорить процесс в десятки раз... Эта идея появилась около 2006 года и стала важной частью революции глубокого обучения.

Но, конечно, содержание революции глубоких сетей не ограничивается тем, что компьютеры стали быстрее, а данных стало больше. Если бы это было так, можно было бы никаких книг не писать, а закончить разговор прямо сейчас. Техническое развитие позволило не только вернуться к идеям 80–90-х годов XX века, но и придумать много новых идей.

Развивались и общие методы обучения нейронных сетей (о них мы поговорим в главе 4 и вообще в первой части книги), и классические архитектуры нейронных сетей, которым в основном посвящена вторая часть, — сверточные сети и автокодировщики (глава 5), рекуррентные сети (глава 6). Но появлялись и совершенно новые архитектуры: порождающие состязательные сети (глава 8) сумели превратить нейронные сети в порождающие модели, нейронные сети в обучении с подкреплением (глава 9) привели к невиданным ранее прорывам, нейробайесовские методы (глава 10) соединили нейронные сети и классический вероятностный вывод с помощью вариационных приближений, а нужды конкретных приложений привели в разработке таких новых архитектур, как сети с вниманием (раздел 8.1) и сети с памятью, которые уже находят и другие применения.

Прежде чем двигаться дальше, скажем пару слов об основных источниках. Книг по обучению глубоких сетей пока не так уж много. Некоторые считают, что эта область еще не вполне устоялась и систематизировать ее пока не обязательно. Как вы наверняка догадались, мы с этим не вполне согласны. Фактически сейчас есть только две известные книги об обучении глубоких сетей: в 2014 году вышла книга сотрудников Microsoft Research Ли Денга (Li Deng) и Донга Ю (Yu Dong) *Deep Learning: Methods and Applications* [117], в которой основной упор делается на приложения для обработки сигналов, то есть для анализа изображений и речи. А совсем недавно, в конце 2016 года, вышла книга Иэна Гудфеллоу (Ian Goodfellow), Йошуа Бенджи и Аарона Курвилля (Aaron Courville) *Deep Learning* [184], в которой дается подробное и идейное введение в тему. Хотя эта книга, формально говоря, наш прямой конкурент, мы ее всячески рекомендуем; кстати, русский перевод

[1] Гордон Мур (Gordon Moore) сформулировал свой закон, по которому число транзисторов в интегральной схеме удваивается примерно каждые два года, а общая производительность систем — примерно каждые 18 месяцев, еще в 1965 году (точнее, тогда он предсказывал удвоение каждый год, а в 1975-м стал более пессимистичным). Конечно, это не закон природы, а просто эмпирическая закономерность, но поразительно, насколько долго эта экспоненциальная зависимость сохраняет силу: закон Мура в наше время (2016–2017 годы) лишь немного замедлился, с двух лет до двух с половиной. Правда, сейчас речь идет уже не только о числе транзисторов в одной схеме, но и о числе ядер в процессоре.

тоже уже существует. Кроме того, есть целый ряд важных больших обзоров, которые до статуса книг не дотягивают, но дают очень широкое представление о глубоком обучении: ранний обзор Бенджи [38] показывает состояние дел на 2009 год, его же более поздний обзор рассматривает глубокое обучение как обучение *представлений* (representation learning) [39], обзор Ли Денга сделан с уклоном в обработку сигналов [115], а обзор Юргена Шмидхубера (Jurgen Schmidhuber) [475] дает очень подробный анализ истории основных идей глубокого обучения.

Есть, конечно, и более специальные обзоры для конкретных областей: обработки изображений [107], обработки естественных языков [182] и многих других. Из статей, в которых речь идет об обучении глубоких сетей «вообще», можно выделить программную статью в Nature трех главных представителей глубокого обучения: Джеффри Хинтона, Йошуа Бенджи и Яна ЛеКуна (Yann LeCun) [314]. На более специализированные источники мы будем ссылаться на протяжении всей этой книги: область молодая, развивается она очень быстро, и в списке литературы, в который мы старались помещать только действительно важные статьи, источников уже накопилось немало. Кстати, следить за последними новостями глубокого обучения лучше всего на *arXiv*[1]: новые работы по обучению глубоких сетей появляются там постоянно, и если вы не заходили на arXiv два-три месяца, может оказаться, что в вашей области уже произошла какая-нибудь очередная мини-революция, а вы и не заметили...

В этой книге мы поговорим о разных архитектурах нейронных сетей, разберем, как именно обучать нейронные сети, какие задачи можно ими решать. Более того, мы дойдем и до практики: почти каждый наш рассказ будет сопровождаться вполне рабочими примерами кода с использованием современных библиотек для обучения нейронных сетей, главным образом TensorFlow и Keras. Подобные библиотеки сделали обучение нейронных сетей общедоступным и стали важной компонентой взрыва интереса и быстрого развития нашей области. Нас ждет очень много интересного. А в этой, вводной, главе мы начнем с того, что поговорим о том, что вообще такое искусственный интеллект и машинное обучение, как они появились, как развивались и как искусственные нейронные сети, которые мы рассматриваем в этой книге, связаны с естественными, с помощью которых вы эту книгу читаете.

---

[1] Хранилище научных препринтов arXiv, расположенное по адресу http://arxiv.org, содержит статьи по многим научным дисциплинам. Но в некоторых областях arXiv стал стандартом де-факто, куда выкладываются практически все серьезные статьи, обычно еще до официальной публикации. К таким областям относится и машинное обучение, в частности обучение глубоких сетей. Правда, стоит отметить, что на arXiv нет никакого рецензирования, так что совсем свежие статьи там могут оказаться и с ошибками, иногда важными. Например, в 2010 году немало шума наделало опубликованное на arXiv решение проблемы $P \neq NP$, предложенное Винаем Деодаликаром. К сожалению, тревога оказалась ложной, и в решении нашлись ошибки. А в 2017-м появилось решение Норберта Блюма, и, когда мы пишем эти слова, решение еще не опровергнуто — впрочем, шансов мало: Александр Разборов уже указал соображения, по которым это доказательство не должно работать... А недавняя инициатива https://openreview.net/ делает вещи, для «классической науки» совершенно немыслимые: там публикуются статьи, еще только поданные на конференции, и рецензии на них.

1.2. Искусственный интеллект и машинное обучение

До чего дошел прогресс — труд физический исчез,
Да и умственный заменит механический процесс.
Позабыты хлопоты, остановлен бег,
Вкалывают роботы, а не человек.

Ю. Энтин. Из к/ф «Приключения Электроника»

Идея искусственного интеллекта давно занимала людей. Гефест создавал роботов-андроидов, как для себя в качестве помощников, так и по заказу; например, построенного Гефестом гигантского человекоподобного робота Талоса Зевс позже подарил царю Миносу для охраны Крита.

Уже в греческих мифах искусственный интеллект мог решать задачи, звучащие вполне современно: Талос трижды в день обегал весь остров, автоматически распознавал среди прибывающих кораблей недружелюбные и бросал в них огромные камни. Примерно тогда же Афродита оживила Галатею, созданную Пигмалионом из мрамора, а еще раньше Иегова и Аллах вдохнули жизнь, самосознание и изрядные когнитивные способности в куски глины.

В иудейской традиции, кстати, особо мудрые раввины могли и сами создавать *големов* — великанов, которых сначала нужно было построить в виде статуи из глины и крови, а затем оживить подходящим артефактом или заклинанием. Големы могли выполнять команды своего создателя, то есть умели распознавать речь и обрабатывать естественный язык. Но синтезировать речь не получалось: если бы голем мог разговаривать самостоятельно, это значило бы, что у него появилась душа, а душу не может вложить даже самый мудрый раввин, — это прерогатива Господа. Поэтому, когда известный алхимик Альберт Великий изготовил искусственную говорящую голову, он очень расстроил своего учителя Фому Аквинского, который, видимо, по этому вопросу был согласен с иудейскими источниками.

Искусственный интеллект давно применялся и к играм: шахматный автомат «Турок» обыграл даже самого Наполеона I; впрочем, здесь быстро выяснилось, что искусственный интеллект не такой уж искусственный...[1] А в наше просвещенное время он стал важной литературной темой практически одновременно с появлением научной фантастики как таковой: начиная с доктора Франкенштейна, идея создания тех или иных мыслящих существ в литературе появляется постоянно.

Считается, что искусственный интеллект как наука начался с *теста Тьюринга*. Формулировка теста впервые появилась в знаменитой статье *Computing Machinery*

[1] В наши дни в честь знаменитого автомата называется сервис Amazon Mechanical Turk, на котором живые люди выполняют небольшие и дешевые задания. Часто «туркеры» как раз и занимаются разметкой наборов данных для моделей, о которых мы будем говорить в этой книге.

and Intelligence, которую Алан Тьюринг[1] выпустил в 1950 году [543]. Впрочем, стоит отметить, что возможность создания «мыслящих машин» и наличия интеллекта у компьютеров обсуждалась и самим Тьюрингом, и его коллегами к тому времени уже как минимум лет десять; это была частая тема для дискуссий в английском Ratio Club, к которому принадлежал и Алан. Наверное, многие читатели слышали основную канву теста Тьюринга: чтобы пройти тест, компьютер должен успешно выдать себя за человека в письменном диалоге между судьей, человеком и компьютером. Иначе говоря, человекоподобных андроидов строить не нужно, но компьютер должен стать неотличим от человека во владении естественным языком.

Любопытно, что исходная формулировка теста Тьюринга была несколько тоньше и интереснее. Действительно, очевидно, что этот тест задуман крайне несправедливо по отношению к несчастным компьютерным программам. Если представить себе «обратный тест Тьюринга», в котором человек попробовал бы выдать себя за компьютер, он мгновенно был бы раскрыт вопросом вроде «Сколько будет $723^3/271$?». Тьюринг понимал, что человеку в предложенной им схеме достаточно просто быть собой, а компьютеру надо выдавать себя за кого-то другого. Поэтому исходная формулировка была основана на популярной тогда имитационной игре[2], в которой мужчина и женщина письменно общаются с судьей. Задача мужчины — выдать себя за женщину, задача женщины — помочь судье правильно разобраться, кто есть кто. Тьюринг предложил сравнивать в этой игре результаты компьютера и живых мужчин: тогда обе стороны вынуждены будут имитировать кого-то третьего. Сам Тьюринг считал, что к 2000 году компьютеры с гигабайтом памяти смогут играть в имитационную игру так, чтобы убеждать человека в 30 % случаев (наилучшим результатом было бы 50 %, то есть в идеале решения судьи были бы неотличимы от бросания честной монетки).

Тест Тьюринга помогает понять, сколько всего нужно сделать, чтобы суметь сконструировать искусственный интеллект: здесь и обработка естественного языка, и представление знаний, и умение делать выводы из полученных знаний, в том числе и обучение на опыте. Однако тест Тьюринга сейчас практически не считается истинным тестом на то, являются ли машины мыслящими. Буквальная формулировка теста породила в наше время достаточно широко известную, но на самом деле

---

[1] *Алан Тьюринг* (Alan Turing, 1912–1954) — английский математик и информатик. Тьюринг был одним из создателей современной теоретической информатики и теории вычислимости: он разработал конструкцию машины Тьюринга и доказал первые результаты о (не)вычислимости. Во время Второй мировой войны Тьюринг был одним из ведущих криптоаналитиков Блетчли-парка и расшифровал ряд важнейших сообщений, закодированных знаменитыми машинами «Энигма». Статьи Тьюринга о математике криптоанализа были рассекречены только в 2012 году. Тьюринг также стал основателем искусственного интеллекта, сформулировав основные положения AI как науки, в частности, знаменитый тест Тьюринга. К сожалению, Тьюринг погиб, не дожив до 42-го дня рождения; мы до сих пор не знаем, было ли это самоубийством из-за назначенного Тьюрингу гормонального лечения гомосексуализма (увы, времена были отнюдь не толерантные) или цианид попал в пресловутое яблоко случайно (это, как ни странно, вполне возможно).

[2] *The Imitation Game* — именно так называется и недавний биографический фильм о Тьюринге.

не слишком научную деятельность по созданию так называемых *чатботов*, нацеленных на поддержание разговора с человеком[1]. Одним из первых и самых известных таких ботов была ELIZA [563], которая еще в 60-е годы XX века могла вести беседу в стиле классического психоаналитика. А в 2014 году появилось (достаточно спорное) сообщение о том, что тест Тьюринга успешно прошел «Женя Густман» (Eugene Goostman), чатбот, созданный тремя русскоязычными программистами. «Женя» представляется собеседникам 13-летним мальчиком из Одессы, и люди часто списывают на это ошибки в английском языке, недопонимания и недостаток знаний. Но чатботы никогда даже не претендовали на то, чтобы «действительно понимать» человеческий язык со всеми контекстами.

А в философском контексте это смыкается с известной конструкцией так называемой *китайской комнаты* Джона Сёрля [477]: представьте себе комнату, полную бумажек со странными символами. На ее стенах записан очень сложный алгоритм перекладывания бумажек, и находящийся в комнате человек перекладывает бумажки согласно этому алгоритму. Оказывается, что алгоритм, к примеру, поддерживает беседу на китайском языке, получая на вход реплики и выдавая ответы на них. Но человек не знает китайского и не понимает входов и выходов, он просто перекладывает бумажки. Кто или что «знает китайский» в этом примере? Можно ли сказать, что комната с бумажками начала «обладать сознанием»?..

Но тест Тьюринга — это только одна из идей постановки конкретной задачи. А вот с самой наукой об искусственном интеллекте произошел достаточно редкий случай: мы, пожалуй, можем проследить точное время и место рождения этой области. В 1956 году четыре отца-основателя искусственного интеллекта — Джон Маккарти (John McCarthy), Марвин Минский (Marvin Minsky), Натаниэль Рочестер (Nathaniel Rochester) и Клод Шеннон (Claude Shannon) — организовали *Дартмутский семинар*, знаменитую летнюю школу в Дартмуте. Заявка на проведение этого семинара была, пожалуй, самой амбициозной грантозаявкой в истории информатики. Вот посмотрите, что писал там Джон Маккарти [431]: «Мы предлагаем исследование искусственного интеллекта сроком на два месяца с участием десяти человек летом 1956 года в Дартмутском колледже, Гановер, Нью-Гемпшир. Исследование основано на предположении, что всякий аспект обучения или любое другое свойство интеллекта может в принципе быть столь точно описано, что машина сможет его имитировать. Мы попытаемся понять, как обучить машины использовать естественные языки, формировать абстракции и концепции, решать задачи, сейчас подвластные только людям, и улучшать самих себя. Мы считаем, что существенное продвижение в одной или более из этих проблем вполне возможно, если специально подобранная группа ученых будет работать над этим в течение лета».

Впечатляет, правда? Не зря 50-е годы XX века — это золотой век классической научной фантастики, когда свои лучшие произведения создавали Айзек Азимов, Рэй Брэдбери, Артур Кларк, Роберт Хайнлайн и многие другие. К этому времени

[1] Сейчас для разработки чатботов и диалоговых систем тоже применяют глубокие нейронные сети — об этом мы вкратце поговорим в разделе 8.1.

относятся и знаменитые три закона робототехники Айзека Азимова, которые сами по себе звучат очень нечетко и противоречиво; понять и принять их к исполнению, пожалуй, даже сложнее, чем пройти тест Тьюринга (многие книги Азимова о роботах как раз на таких внутренних противоречиях и нечеткостях и основаны). Самое удивительное в этой истории состоит в том, что заявка все-таки была удовлетворена, Дартмутский семинар был проведен, а тот факт, что заявленных целей за эти два месяца достичь не удалось, не послужил основой для немедленных оргвыводов и не поставил крест на всем искусственном интеллекте.

Началось все с исследований, которые продолжали начатую логиками тему *автоматического логического вывода* (automated theorem proving). Первые программы, создававшиеся как шаги на пути к искусственному интеллекту, пытались строить выводы в заданных исследователями формальных системах. Например, появившаяся в 1959 году программа с амбициозным названием «Универсальный решатель задач» (General Problem Solver, G.P.S.) [392] могла строить вывод в системах, заданных логическими формулами определенного вида[1]. Другие программы того времени пытались оперировать более ограниченными предметными областями, так называемыми микромирами (microworlds): решать словесные алгебраические задачи (те, где из бассейна вытекают навстречу друг другу два поезда со скоростью 40 км/ч каждый), переставлять геометрические фигуры в трехмерном пространстве и т. д. Вся эта наука тогда обычно называлась *кибернетикой* вслед за книгой Норберта Винера «Кибернетика, или Управление и связь в животном и машине» [568].[2]

Тогда же, во второй половине 1950-х и начале 1960-х годов, появились и самообучающиеся машины, в частности перцептрон Розенблатта, о котором мы будем много говорить в главе 3. Они тут же получили широкую огласку, и людям начало казаться, что до реализации законов робототехники уже рукой подать. Но такие

---

[1] Для читателей, прослушавших курс математической логики, уточним: формулами, состоящими из хорновских дизъюнктов. Означенные читатели поймут, что это как раз те формулы, вывод для которых автоматически построить вполне реально... но только по сравнению с более сложными теориями, на самом деле вычислительно это все равно весьма сложная процедура, быстро приводящая к комбинаторному взрыву и экспоненциальному перебору вариантов.

[2] Широко известно, что в начале 1950-х годов в СССР прошла массовая пропагандистская акция против кибернетики в понимании Винера. Дело было поставлено со сталинским размахом, даже в «Философский словарь» 1954 года издания попало определение кибернетики как «реакционной лженауки». Однако быстро стало понятно, что компьютеры — это всерьез и надолго, и уже в 1955 году выдающиеся математики Соболев (тот самый, Сергей Львович), Китов и Ляпунов (не Александр Михайлович, конечно, а Алексей Андреевич, один из основоположников советской кибернетики) писали в «Вопросах философии» так: «Некоторые наши философы допустили серьезную ошибку: не разобравшись в существе вопросов, они стали отрицать значение нового направления в науке... из-за того, что некоторые невежественные буржуазные журналисты занялись рекламой и дешевыми спекуляциями вокруг кибернетики... Не исключена возможность, что усиленное реакционное, идеалистическое толкование кибернетики в популярной реакционной литературе было специально организовано с целью дезориентации советских ученых и инженеров, с тем, чтобы затормозить развитие нового важного научного направления в нашей стране». Как видите, разворот на 180 градусов произошел быстро, и советские алгоритмисты в общем-то всегда оставались на переднем крае науки.

авансы искусственный интеллект тогда оправдать не мог. В разделе 3.2 мы еще поговорим о том, как началась первая «зима искусственного интеллекта». Одной из причин стал полный провал большого проекта по машинному переводу, который адекватно сделать в те годы было невозможно, а другой — появившееся понимание того, что одним перцептроном много не сделать.

Поэтому 1970-е годы стали временем расцвета *систем, основанных на знаниях* (knowledge-based systems), которые до сих пор являются важной частью науки об *экспертных системах*. Суть здесь состоит в том, чтобы накопить достаточно большой набор правил и знаний о предметной области, а затем делать выводы. Одним из самых ярких примеров таких проектов стала система MYCIN, посвященная идентификации бактерий, вызывающих серьезные инфекции, а затем рекомендующая антибиотики [65, 494]. В ней было около 600 правил, и ее результаты (она выдавала подходящий метод лечения в 69 % случаев) были не хуже, чем у опытных врачей, и существенно лучше, чем у начинающих. Более того, такие системы могли объяснить, как именно они пришли к тому или иному решению, и оценить свою уверенность в этой гипотезе. Позднее они стали прообразами графических вероятностных моделей.

Затем исследователи снова вспомнили о нейронных сетях: к началу 1980-х годов был разработан алгоритм обратного распространения ошибки (его мы подробно обсудим в главе 2), что открыло дорогу самым разнообразным архитектурам. Одним из ключевых понятий 1980-х годов был *коннекционизм* (connectionism): пришедшая из когнитивных наук идея о том, что нужно не пытаться задавать аксиоматические формальные системы для последующих рассуждений в них, а строить большие ансамбли параллельно работающих нейронов, которые, подобно человеческому мозгу, как-нибудь сами разберутся, что им делать. К концу восьмидесятых уже появилась большая часть основных архитектур, о которых мы будем говорить в этой книге: сверточные сети, автокодировщики, рекуррентные сети...

А в целом в искусственном интеллекте начались первые коммерческие применения. Рассказывают, что один из первых AI-отделов, появившийся в компании DEC (Digital Equipment Corporation), к 1986 году экономил компании около 10 миллионов долларов в год, очень серьезную по тем временам сумму. Однако и здесь исследователи и особенно «стартаперы» восьмидесятых не удержались. На волне всеобщего увлечения искусственным интеллектом многие компании снова начали направо и налево раздавать обещания. Поэтому вторая волна увлечения искусственным интеллектом закончилась в начале девяностых, когда многие компании не смогли оправдать завышенных ожиданий и лопнули[1].

В 1990-е годы основной акцент сместился на машинное обучение и поиск закономерностей в данных, причем нейронные сети, как мы уже упоминали выше, не

[1] Вероятно, именно с тех пор словосочетание «искусственный интеллект», по крайней мере в российской науке, пользуется не слишком доброй славой; многие математики недовольно поморщатся, если услышат, что вы занимаетесь искусственным интеллектом. Так что советуем этого словосочетания не употреблять, а говорить «машинное обучение»: так и точнее, и безопаснее.

считались особенно перспективными. Зато самих данных, особенно с развитием Интернета, становилось все больше, компьютеры становились все быстрее. В итоге в середине 2000-х годов очередная новая идея наконец-то сработала, к ней быстро подтянулись другие, и все, как говорится, заверте. Этому посвящена вся наша книга, так что не станем пытаться вкратце повторить всю историю сейчас.

Но что же все-таки с искусственным интеллектом? Как сейчас поживает извечная мечта человечества? Конечно, сейчас уже никто не обещает, что мы вот-вот построим искусственный интеллект, и изящные андроиды вот-вот будут приносить нам кофе, отвечать на звонки, выносить мусор и выполнять все прочие, менее невинные команды.

Однако современные исследователи и футурологи, совсем как в 1960-е годы, снова весьма оптимистично настроены по поводу искусственного интеллекта. Заметная часть специалистов считает, что сильный искусственный интеллект (strong AI, то есть искусственный интеллект человеческого уровня или выше) вполне может быть создан еще при нашей жизни [380]. И это приводит к куда менее оптимистичным рассуждениям о том, чем нам, людям, грозит такое развитие событий.

Нет, речь не идет о том, что искусственный интеллект заменит людей и приведет к тому, что нам с вами нечего будет делать и придется всю жизнь бесцельно сибаритствовать или искать новый, непродуктивный смысл жизни: этот сценарий звучит явно недостаточно катастрофически. Скорее речь идет о том, что улучшающий себя искусственный интеллект, чью цель опрометчиво определили как производство канцелярских скрепок, может для достижения этой цели ненавязчиво захватить власть над планетой и превратить ее целиком в фабрики по производству скрепок и космических кораблей, предназначенных для превращения в скрепки остатка видимой Вселенной... В общем, очень интересные рассуждения. Основные источники на эту тему принадлежат перу шведского философа Ника Бострома (Nick Bostrom), автора известной книги «Искусственный интеллект: этапы, угрозы, стратегии» [52], и американского исследователя искусственного интеллекта (не в математическом, а скорее в философском смысле) Элиэзера Юдковского (Eliezer S. Yudkowsky), одного из сооснователей Института исследований машинного интеллекта (Machine Intelligence Research Institute, MIRI), автора ряда книг и обзоров об угрозах искусственного интеллекта [53, 206, 578, 579], а также обширного и очень интересного популярного изложения принципов рационального мышления [581][1].

Но это пока дело будущего. А сейчас пора переходить к настоящему: к тому, чем сегодня занимается машинное обучение, какие задачи перед собой ставит и какие методы использует. Начнем с очень краткого обзора всей области в целом.

---

[1] Впрочем, если вы хотите прочитать *действительно* популярное изложение этих принципов, не проходите мимо книги Элиэзера Юдковского «Гарри Поттер и принципы рационального мышления» (Harry Potter and the Methods of Rationality) [580]. Это, конечно, не высокая литература, но идеи изложены очень захватывающе.

1.3. Немного о словах: каким бывает машинное обучение

> Аристотель тоже везде ищет истину. Но как? Только путем бесконечного расчленения понятий и путем выяснения тончайшей терминологии, заставляющей иной раз переходить к самому настоящему словарю весьма дробной и утонченной терминологии.
>
> *А. Ф. Лосев. История античной философии в конспективном изложении*

> И пусть нас не смущает то, что части сущностей находятся в целых как в подлежащих, чтобы нам не пришлось когда-нибудь утверждать, что эти части не сущности: ведь о том, что находится в подлежащем, было сказано, что оно находится в нем не так, как части содержатся в каком-нибудь целом.
>
> *Аристотель*

В дальнейшем в книге вы встретите много разных слов, которые могут оказаться для вас новыми. Если вы еще не владеете этой терминологией, не пугайтесь, обычно все не так уж сложно. Но часть терминологии, так сказать, общую канву и структуру машинного обучения, стоит обсудить заранее.

Начнем мы с того, что такое, собственно, *машинное обучение* (machine learning). Интуитивно понятно, что «обучение» — это когда некая модель каким-то образом «обучается», а потом начинает выдавать результаты, то есть, скорее всего, что-то предсказывать. Можно даже дать очень общее определение «обучаемости», примерно такое, какое дает Томас Митчелл в классической книге «Машинное обучение» [368]: «Компьютерная программа *обучается* по мере накопления опыта относительно некоторого класса задач T и целевой функции P, если качество решения этих задач (относительно P) улучшается с получением нового опыта».

Хотя это определение звучит чрезвычайно обобщенно и абстрактно, оно на самом деле позволяет прояснить некоторые важные моменты. Например, центральное место в нем занимают не данные (хотя они тоже есть), а целевая функция. Когда вы начинаете решать любую практическую задачу, крайне важно еще «на берегу» определить целевую функцию, договориться о том, как вы будете оценивать результаты. Выбор целевой функции полностью определяет всю дальнейшую работу, и даже в похожих задачах разные целевые функции могут привести к совершенно разным моделям. Например, было бы здорово «научить компьютер читать», но сначала нужно определить, что это, собственно, значит. Уметь правильно отвечать на вопросы по «прочитанному» тексту? Сделать синтаксический разбор

Рис. 1.1. Общая классификация постановок задач машинного обучения

предложения? Показать самые релевантные данному тексту статьи «Википедии»? Разные ответы приводят к разным моделям и направлениям исследований.

Но все-таки определения Митчелла нам будет недостаточно. Какие бывают задачи машинного обучения, из чего оно состоит? Мы показали основную классификацию задач машинного обучения на рис. 1.1. Два основных класса задач машинного обучения — это задачи *обучения с учителем* (supervised learning) и *обучения без учителя* (unsupervised learning).

При обучении с учителем на вход подается набор тренировочных примеров, который обычно называют *обучающим* или *тренировочным набором данных* (training set или training sample — тренировочная выборка), и задача состоит в том, чтобы продолжить уже известные ответы на новый опыт, выраженный обычно в виде *тестового набора* данных (test set, test sample). Основное предположение здесь в том, что данные, доступные для обучения, будут чем-то *похожи* на данные, на которых потом придется применять обученную модель, иначе никакое обобщение будет невозможно. Для «чтения» текста пример обучения с учителем — это обучение модели, которая строит деревья синтаксического разбора предложений (какие слова от каких зависят) по набору деревьев, построенных людьми для конкретных предложений. Предположение здесь в том, что деревья разбора строятся по одним и тем же законам, и модель, обученную на некотором наборе деревьев разбора, можно будет применить и к новым предложениям, не входящим в обучающий набор. Если это предположение нарушится, модель работать не будет. Например, если лингвисты размечали предложения на английском языке, а потом применили обученную модель к немецкому, где буквы примерно те же, но синтаксис совершенно другой, ожидать от модели разумного поведения не стоит.

Задачи обучения с учителем обычно делятся на задачи *классификации* и *регрессии*. В задаче классификации нужно поданный на вход объект определить в один из (обычно конечного числа) классов, например, разделить фотографии животных на кошек, собак, лошадей и «все остальное»; или по фотографии человеческого лица понять, кто из ваших друзей в социальной сети на ней изображен. Если продолжать пример с языком, то типичная задача классификации — это разметка слов по частям речи. А в задаче регрессии нужно предсказать значение некоей функции, у которой обычно может быть бесконечно много разных значений. Например, по росту человека предсказать его вес, сделать прогноз завтрашней погоды, предсказать цену акции или, скажем, выделить на фотографии прямоугольник, в котором находится человеческое лицо — сделать это необходимо, чтобы затем эти прямоугольники подать на вход упомянутому выше классификатору.

Деление на регрессию и классификацию, конечно, очень условно, можно легко придумать «промежуточные» примеры (тот же разбор предложения — к какому классу отнести задачу «построить дерево»?). Но обычно ясно, какую задачу мы решаем, и это деление имеет содержательный смысл: меняются целевые функции и, как следствие, процесс обучения. А есть и откровенно иные виды задач, не укладывающиеся в эту несложную классификацию. Например, в поисковых и рекомендательных системах часто встречается задача *обучения ранжирования* (learning to rank). Она ставится так: по имеющимся данным (в поисковой системе это будут тексты документов и прошлое поведение пользователей) отранжировать, расставить имеющиеся объекты в порядке убывания целевой функции (в поисковой системе она называется *релевантностью*: насколько данный документ подходит, чтобы выдать его в ответ на данный запрос). Эта задача чем-то похожа на задачу регрессии — нам так или иначе нужно предсказывать непрерывную целевую функцию, ту самую релевантность. Но при этом значений самой функции в данных совсем нет, да они нам и не важны. Имеют значение только результаты сравнения этой функции на разных объектах (какой документ будет выше другого в поисковой выдаче). Это приводит к ряду интересных и специфических методов обучения.

Если же размеченного набора данных, соответствующего конкретной задаче, нет, а есть просто данные, в которых надо «найти какой-нибудь смысл», то возникают задачи обучения без учителя. Типичнейший пример задачи обучения без учителя — это *кластеризация* (clustering): нужно разбить данные на заранее неизвестные классы по некоторой мере похожести так, чтобы точки, отнесенные к одному и тому же кластеру, были как можно ближе друг к другу, как можно более похожи, а точки из разных кластеров были бы как можно дальше друг от друга, как можно менее похожи. Например, решив задачу кластеризации, можно выделить семейства генов из последовательностей нуклеотидов, или кластеризовать пользователей вашего веб-сайта и персонализовать его под каждый кластер, или сегментировать медицинский снимок, чтобы легко было понять, где же там опухоль.

Еще одна часто встречающаяся задача обучения без учителя — это *снижение размерности* (dimensionality reduction), когда входные данные имеют большую

размерность (например, если у вас на входе разбитый на слова текст, размерность будет исчисляться десятками тысяч, а если фотографии — миллионами), а задача состоит в том, чтобы построить представление данных меньшей размерности, которое тем не менее будет достаточно полно отражать исходные данные. То есть, например, по представлению меньшей размерности можно будет достаточно успешно реконструировать исходные точки большой размерности. Это можно рассматривать как частный случай общей задачи *выделения признаков* (feature extraction), и мы будем подробно говорить об этом на протяжении всей книги, а особенно в разделе 5.5, где речь пойдет об автокодировщиках.

И наконец, третий и самый общий класс задач обучения без учителя — задача *оценки плотности*: нам даны точки данных $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ и, возможно, какие-то априорные представления о том, откуда взялись эти точки, а хочется оценить распределение $p(\boldsymbol{x})$, из которого они получились. Это очень общая постановка задачи, к ней можно многое свести, и нейронные сети тоже отчасти ее и решают.

Нередко в жизни возникает нечто среднее между обучением с учителем и обучением без учителя. Так обычно получается тогда, когда неразмеченные примеры найти очень легко, а размеченные получить сложно. Например, во все том же примере с синтаксическим разбором набрать сколько угодно неразмеченных текстов не представляет никакой сложности, а вот вручную нарисовать даже одно дерево нелегко. Или, скажем, задумайтесь о том, что такое «размеченные данные» для распознавания речи. Просто установить соответствие между звуковым файлом с речью и текстом, особенно если тексты достаточно длинные, здесь может быть недостаточно. По-настоящему размеченные данные для распознавания — это звуковые файлы, в которых вручную отмечены (или хотя бы проверены) границы *каждой фонемы*, каждого звука человеческой речи. Это адский труд, обычно делегируемый младшим научным сотрудникам, но даже целая армия лингвистов-фонологов будет двигаться достаточно медленно. А неразмеченных звуков живой человеческой речи вы можете записать сколько угодно, просто включив диктофон на запись. Такую ситуацию иногда называют *обучением с частичным привлечением учителя*, или полуконтролируемым обучением (английский термин *semi-supervised learning* устоялся куда больше), и с ней мы тоже не раз столкнемся в этой книге.

В главе 9 мы с вами встретимся с другой постановкой задачи — *обучением с подкреплением* (reinforcement learning), когда агент, находясь в некоей среде, производит те или иные действия и получает за это награды. Цель агента — получить как можно большую награду с течением времени, а для этого неплохо бы понять, какие действия в конечном счете ведут к успеху. Так можно обучиться играть в разные игры или, например, сделать отличный протокол для A/B-тестирования.

И это, конечно же, далеко не все. Но сейчас мы все же прекратим перечислять разные области машинного обучения и будем двигаться к той области, которая нас более всего интересует в этой книге, к нейронным сетям.

А тем читателям, которые хотели бы узнать о машинном обучении больше, мы рекомендуем два уже ставших классическими учебника: книгу Кристофера

Бишопа (Christopher Bishop) *Pattern Recognition and Machine Learning* [44] и книгу Кевина Мерфи (Kevin Murphy) *Machine Learning: A Probabilistic Approach* [381]. В обеих книгах последовательно описывается вероятностный подход к машинному обучению, основанный на теореме Байеса. Это, по нашему мнению, математически наиболее полный и корректный взгляд на происходящее с обучающимися моделями.

Строгий математический подход хотелось бы, конечно, применить и к глубокому обучению, но пока людям это удается только частично[1], а математическая структура тех сложных геометрических конструкций, которые будут у нас получаться и в пространствах признаков, и тем более в пространствах самих объектов для обучения, практически не изучена... Как знать, может быть, вы, читатель, поможете закрыть этот досадный пробел?

1.4. Особенности человеческого мозга

> Чтобы вместить в себя столько чужих мозгов, и к тому же таких великих и мощных, необходимо (как выразилась о ком-то одна девица, первая среди наших принцесс), чтобы собственный мозг потеснился, съежился и сократился в объеме.
>
> *М. Монтень. О педантизме*

Когда мы говорим о задачах машинного обучения, мы быстро сталкиваемся с задачами, с решением которых человеческий мозг до сих пор справляется[2] лучше и быстрее, чем компьютер. Например, мы лучше обращаемся с естественным языком: можем прочитать, понять и содержательно изучить книгу, а современные компьютеры испытывают большие проблемы с ответами даже на очевидные для человека вопросы. Да и вообще мы хорошо умеем обучаться в широком смысле этого слова — компьютерные программы пока очень далеки от человеческого уровня обобщений и поиска взаимосвязей между разнородной информацией. Что же делает человеческий мозг таким эффективным? Как он умудряется достичь таких высот? В чем разница между тем, что делают нейроны в мозге, и тем, что делают транзисторы в процессоре? Тема эта неисчерпаема, поэтому здесь мы приведем только пару локальных примеров, с помощью которых продемонстрируем основные различия и отчасти мотивируем отдельные особенности нейронных сетей.

Человеческий (да и любой другой) мозг состоит из *нейронов*. У каждого из них есть один длинный отросток, *аксон*, и много коротких отростков, *дендритов*,

---

[1] О нейробайесовских подходах мы поговорим в главе 10, но там подход будет противоположным: мы будем по-прежнему использовать нейронные сети как «черный ящик», но они начнут помогать нам делать байесовский вывод и приближать сложные распределения.

[2] Или до недавнего времени справлялся: прогресс все ускоряется, сингулярность близко, как знать, может быть, когда вы это читаете, все уже решено...

которые связываются с аксонами других нейронов[1]. Связи между дендритами и аксонами тоже имеют сложную структуру, в которую мы не будем вдаваться, и называются *синапсами*. Связей в мозге очень много: порядка 10^{11} (сто миллиардов) нейронов, у каждого из которых в среднем 7000 связей, то есть получается, что в нашем мозге содержится порядка 10^{14}–10^{15} (сто триллионов) синапсов.

Каждый нейрон время от времени посылает по аксону нервный импульс (по-английски он называется spike), который имеет электрическую природу. Пока нейрон жив, он никогда не останавливается и продолжает подавать сигналы. Но при этом нейрон может находиться в двух разных состояниях: когда он находится в «выключенном» состоянии, частота подачи сигналов маленькая, а когда он возбуждается, «включается», частота подачи импульсов (firing rate) сильно увеличивается. Активация нейронов зависит от сигналов, приходящих через синапсы и затем дендриты от других нейронов. Связь в синапсе, кстати, может быть как положительная (excitation), когда активация соседнего нейрона повышает вероятность активации нашего, так и отрицательная (inhibition), когда активация соседнего нейрона, наоборот, подавляет активность.

Хотелось бы, конечно, провести аналогию между нейронами и транзисторами, из которых состоит процессор, но уже на уровне подачи нервных импульсов мы видим первое важное различие между мозгом и компьютером. Дело в том, что нейрон работает стохастически: он выдает электрические сигналы через случайные промежутки времени. Их последовательность можно довольно хорошо приблизить пуассоновским случайным процессом, интенсивность которого меняется в зависимости от того, возбужден нейрон или нет. В компьютерах тоже есть гейты, обменивающиеся сигналами друг с другом, но они делают это совершенно не стохастически, а с очень жесткой синхронизацией: частота процессора, давно уже измеряющаяся в гигагерцах, — это и есть частота такой синхронизации. На каждом такте гейт одного уровня передает сигнал следующему уровню, и делают они это хоть и несколько миллиардов раз в секунду, но строго одновременно, по команде.

Возможно, дело в том, что в биологии все нечетко, и синхронизация между нейронами была бы желательна, но достичь ее у создавшей нас эволюции просто не получилось? Вовсе нет: простейшие наблюдения показывают, что на самом деле нейроны хорошо умеют синхронизироваться и очень точно засекать весьма короткие промежутки времени.

Самая простая и яркая иллюстрация этого — стереозвук. Когда вы переходите от одной стороны комнаты к другой, вы можете, опираясь исключительно на звук, идущий из колонок вашего компьютера, определить направление на его источник. Очевидно, в доисторические времена для людей было крайне важно понять, слева

[1] На самом деле это только один возможный тип нейронов. Бывают нейроны без аксона вовсе, бывают с одним аксоном и без дендритов, бывают с одним дендритом, но все они встречаются в достаточно редких ситуациях, часто не у людей. Наш мозг однозначно состоит именно из биполярных нейронов, у которых один аксон и много дендритов.

или справа хрустнула ветка под лапой тигра... Чтобы узнать направление, вы отмечаете разницу во времени, когда звук приходит в левое и правое ухо. Расстояние между внутренними ушами не слишком большое, сантиметров двадцать. И если разделить его на скорость звука (340 м/с), получится очень короткий интервал, десятые доли миллисекунды, который, тем не менее, нейроны отлично распознают, что позволяет определить направление с хорошей точностью. То есть ваш мозг в принципе мог бы работать как компьютер с частотой, измеряющейся килогерцами. С учетом огромной степени параллелизации, достигнутой в архитектуре мозга, это могло бы привести к вполне разумной вычислительной мощности. Но почему-то мозг этого не делает.

Кстати, о параллелизации. Второе важное замечание про работу человеческого мозга: мы распознаем лицо человека за пару сотен миллисекунд. А частота импульсов в аксонах бывает от 10 Гц в неактивном состоянии до примерно 200 Гц во время самой сильной активации. Это значит, что связи между отдельными нейронами активируются минимум за десятки миллисекунд, и в полном цикле распознавания человеческого лица не может быть последовательной цепочки активаций длиннее, чем буквально несколько нейронов; скорее всего, меньше десятка!

То есть мозг, с одной стороны, содержит огромное число нейронов и еще больше связей между ними, но с другой — устроен очень плоско по сравнению с обычным процессором. Процессор в компьютере исполняет длинные последовательные цепочки команд, обрабатывая их в синхронном режиме, а у мозга цепочки короткие, зато работает он асинхронно (стохастически) и с очень высокой степенью параллелизации: нейроны активируются сразу во многих местах мозга, когда начинают распознавать лицо и делать много других увлекательных вещей. Можно сказать, что с этой точки зрения мозг больше похож на видеокарту, чем на процессор, и к видеокартам мы действительно еще вернемся.

Еще одна важная особенность состоит в том, что аксоны могут быть очень длинными: например, от спинного мозга в конечности идут аксоны длиной около метра. Поэтому один аксон может с легкостью пересечь чуть ли не весь мозг, в связи с чем структура связей между нейронами в мозге чрезвычайно сложная. В главе 5 мы немного поговорим о том, как это устроено в зрительной коре, но модель, в которой нейроны делятся на «уровни» (как в глубокой нейронной сети) и активация аккуратно распространяется от нейронов, связанных с органами чувств, до абстрактных понятий, а затем обратно для активации мышечной ткани, на самом деле сильно упрощена.

В человеческом мозге есть огромное число «горизонтальных» связей между нейронами (когда связаны друг с другом нейроны одного уровня), множество замкнутых цепочек связанных нейронов, а также неожиданных и непонятных связей нейронов из совершенно разных областей мозга. Все эти связи время от времени срабатывают, но единой картины того, зачем все это нужно и как используется, в науке пока не сложилось. Так что на самом деле мы очень плохо понимаем, как работает настоящий человеческий мозг, и искусственные нейронные сети — это не

попытка приблизиться к реальной структуре, а достаточно абстрактные модели, созданные для решения оптимизационных задач.

Здесь стоит отметить, что в нейробиологии и информатике есть целое направление исследований, связанное с моделированием биологических, «настоящих» нейронов. Разных моделей много, их можно условно разделить на электрические, предсказывающие напряжение на клеточной мембране нейрона [480], и естественные, которые прогнозируют вероятность (и частоту) активации как функцию от входного воздействия [395]. Но это направление очень далеко от темы нашей книги, так что углубляться в него мы не станем. Настолько абстрактные модели, как перцептрон, в котором уровень активации нейрона — это нелинейная функция активации от взвешенной суммы входов (мы подробно рассмотрим перцептрон в главе 3, да и во всей книге), не представляют большого интереса для биологов. Например, в них выход нейрона вовсе не зависит от времени, а большинство «приближенных к биологии» моделей стараются имитировать стохастические процессы порождения нервных импульсов. Используются эти модели в наше время не только для интереса, но и для разработки «протезов» нейронов, например, для восстановления сетчатки глаза [422].

Зато очень близко к нашей теме другое важнейшее свойство человеческого мозга — его *пластичность*. Многие исследования мозга, особенно ранние, выделяли в нем зоны, отвечающие за те или иные функции. По большому счету, только это и было доступно первым нейробиологам: они могли изучать, что человек теряет при травмах того или иного отдела мозга. Так, например, *центр Брока*, открытый Полем Брока[1] еще в 1865 году, отвечает за артикуляцию речи. Если центр Брока поврежден, больной по-прежнему все понимает, но при попытках говорить получается ерунда: распадается грамматика, не удается подбирать правильные предлоги, путаются звуки в словах. А зона Вернике отвечает, наоборот, за понимание речи на слух. В человеческом мозге много узко специализированных участков.

Исходя из этих исследований, можно подумать, что мать-природа (точнее, мать-эволюция) создала человеческий мозг подобно компьютеру с детально разработанной спецификацией: есть «видеокарта», отвечающая за зрение, есть «звуковая карта», обрабатывающая звук, есть «чатбот», который обучился языку, и все они представляют собой особые архитектуры нейронов, специально подогнанные под соответствующую задачу еще на генетическом уровне.

Такая система взглядов была, конечно, вполне логична на ранних этапах изучения мозга. Основателем этой теории считается Галилео Галилей, а пользовавшаяся

[1] *Поль Брока* (Paul Pierre Broca, 1824–1880) — французский хирург и антрополог. Брока был фактически основателем современной антропологии: в середине XIX века он провел ряд исследований, сравнивая характеристики скелетов разных времен (пришлось раскапывать могилы, что тогда не очень приветствовалось), основал первое в Европе общество антропологии и журнал «Антропологическое обозрение». Впрочем, в наше время ученому изрядно бы досталось: исходя из того, что размер мозга напрямую связан с уровнем интеллекта, он пришел к выводу, что мужчины в среднем значительно умнее женщин, а также разделил расы на «высшие» и «низшие» на основе своих сравнений размеров мозга. Кстати, мозг самого Поля Брока сейчас находится в «Музее Человека» в Париже.

большой популярностью в XIX веке френология по сути своей основана именно на этом предположении.

Однако это вовсе не обязательно так! Исследования нейропластичности показывают, что нейроны из самых разных областей мозга могут при необходимости принимать на себя функции, обычно им не свойственные. Сейчас уже ясно, что в нашем мозге не только постоянно образуются новые связи между нейронами (новые синапсы), но и происходит непрерывное обновление самих нейронов (нейрогенезис). А уже существующие нейроны могут переобучиться для того, чтобы обрабатывать совершенно новые сигналы. Считается, что на нейропластичности основан феномен фантомных болей: нейроны, которые раньше «отвечали» за утраченные органы, начинают обучаться чему-то новому, а другие отделы мозга по привычке интерпретируют новые сигналы как идущие от уже давно не существующих конечностей.

Более того, уже сейчас разрабатываются очень интересные устройства, основанные на нейропластичности. Система BrainPort, например, пытается обучить человека видеть... языком! Информация от укрепленной на голове камеры поступает — прямо в цифровом виде, пиксел за пикселом, — на специальную матрицу электродов, укрепленную на языке (язык здесь нужен только потому, что это очень чувствительный орган, на котором много нервных окончаний). В результате человек действительно способен обучиться видеть (не так, как обычными глазами, конечно), то есть мозг, по всей видимости, каким-то образом распознает, что осязательная информация, поступающая с языка, больше не похожа на вкус еды, а скорее напоминает зрительные образы, и «перенаправляет» ее в зрительную кору. Да и обычный кохлеарный имплантат, которые уже сегодня помогает сотням тысяч слабослышащих людей, основан на схожем принципе: он не пытается симулировать слуховые ощущения во внутреннем ухе, а оцифровывает звук и подает его на электроды, непосредственно стимулирующие кохлеарный нерв.

И даже еще более того, в этих примерах можно было бы сказать, что у мозга уже «есть аппарат» для обработки изображений, и дообучение могло бы состоять всего лишь в том, чтобы научиться передавать данные из необычной «точки входа» в нужные участки коры головного мозга (собственно, видимо, так оно и происходит), а не в том, чтобы научиться обрабатывать с нуля эти данные. Однако есть примеры обучения и совершенно новым, нехарактерным для человека вещам. Например, некоторые слепые люди развивают в себе способность к *эхолокации*, определению местоположения окружающих объектов по отраженному от них звуку, — совсем как у летучих мышей (ну ладно, не так хорошо, как у летучих мышей, конечно). Некоторые слепые могут просто щелкать языком и определять окружающие объекты, прислушиваясь к отзвукам [527]. А в других экспериментах вполне зрячие люди обучались эхолокации, получая сигналы с укрепленного на них сонара.

Итак, мы узнали, что человеческий мозг состоит из связанных между собой нейронов, которые активируются в зависимости от своих «предков» и при активации передают друг другу сигналы. Мы выяснили, что мозг может адаптироваться к новым источникам информации, даже изначально совершенно чуждым ему. Исследования нейропластичности, да и просто тот факт, что все нейроны в мозге

работают примерно одинаково, убеждают нас в том, что весь мозг функционирует по некоему «единому алгоритму», который может обучать конгломераты из нейронов на самых разных данных, выделяя признаки из потока неструктурированной информации[1]. Мы еще не раз вернемся к тому, как работает настоящий человеческий мозг, а о зрительной коре подробно поговорим в разделе 5.1.

При создании искусственного интеллекта очень естественно задаться вопросом: можем ли мы этот единый алгоритм тоже как-то промоделировать? Очень заманчиво было бы построить модель этого «единого алгоритма», сгруппировать вместе много-много искусственных нейронов и получить готовый к употреблению мозг. Основная идея искусственных нейронных сетей — собрать сеть из простых нейронов, активирующихся или не активирующихся в зависимости от снабженных поддающимися тренировке весами, — действительно позаимствована у природы. Однако дальше путь развития искусственного интеллекта отошел от природы; настоящие нейроны устроены значительно сложнее, чем те модели, которые мы будем обсуждать в этой книге. В разделе 3.4 мы увидим, как устроены модели обучения нейронов, и поговорим о том, почему то, что мы делаем в искусственных нейронных сетях (а значит, и во всей этой книге), не очень-то похоже на то, что делается у нас в голове. А в следующем разделе сделаем небольшое лирическое отступление и поговорим о том, почему достижения нейробиологии на макроуровне пока еще не вполне годятся на роль образцов для подражания.

1.5. Пределы нейробиологии: что мы на самом деле знаем?

> Даже если достигнута полная ясность, то всегда остается еще не известным, насколько точно соответствует эта система понятий реальности.
>
> *В. Гейзенберг. Физика и философия. Часть и целое*

> С русским читателем следует быть ответственнее. Француз прочитает, к примеру, маркиза де Сада, ухмыльнется дико и пойдет дальше. А у русского читателя чердак поедет: вон оно как умные-то люди, оказывается...
>
> *В. Шинкарёв. Митьковские пляски*

Итак, мы выяснили (разумеется, весьма интуитивно), что в мозге, созданном эволюцией, похоже, заложен некий «единый алгоритм», по которому мозг может обучаться. Можно сказать, что одна из далеких, фактически конечных целей программы по созданию искусственного интеллекта — это понять или хотя бы просто

[1] Недавно вышедшая научно-популярная книга о машинном обучении от Педро Домингоса так и называлась: *The Master Algorithm* (в русском переводе — «Верховный алгоритм») [125].

моделировать этот «единый алгоритм» и построить программы, которые могли бы его использовать и обучаться примерно так же, как это делаем мы с вами.

Давайте, однако, немного поиграем в адвоката дьявола. Мы часто в этой книге ссылаемся на утверждения о том, что «мозг работает так» или «мозг работает эдак». Но действительно ли все это похоже на то, что делает человеческий мозг? Конечно, мы не станем специально вводить читателя в заблуждение: если мы пишем что-то о мозге, значит, хотя бы из третьих (но надежных!) рук слышали, что именно таково современное понимание того, как мозг функционирует, что делают отдельные нейроны и как это все комбинируется в тот удивительный объект, который Митио Каку[1] заслуженно называет «самым сложным объектом во Вселенной» [268]. Но насколько можно доверять «современному пониманию»? Мы ведь не можем понять мозг целиком и осознать, что делает каждый нейрон по отдельности, мы можем только проводить какие-то локальные исследования, смотреть, что меняется под влиянием тех или иных внешних воздействий, часто довольно грубых (как, например, травмы, при которых большие фрагменты мозга перестают работать). Как вообще проверить, что наши методы дают что-то похожее на правду?

В своей недавней яркой работе «Может ли нейробиолог понять микропроцессор?»[2] [263] Эрик Джонас и Конрад Кординг пытаются проследить, насколько бы получилось у методов современной нейробиологии проанализировать какой-нибудь очень простой «мозг» на примере процессора MOS 6502. Такие процессоры устанавливались в Apple I и Atari VCS; кстати, к классическим играм Atari мы еще вернемся в разделе 9.3. Сам чип процессора состоит из всего 3510 транзисторов; для исследования была построена точная цифровая реконструкция чипа, которая могла запускать те самые классические игры *Donkey Kong* и *Space Invaders* и была способна порождать для дальнейшего анализа около 1,5 Гбайт данных за секунду эмуляции. Если сравнить это с огромными возможностями человеческого мозга, видно, что задачу перед собой ученые поставили гораздо менее амбициозную, чем изучение настоящего мозга, а полноценные данные о состоянии каждого транзистора после каждого такта — это такой уровень детализации, о котором современная нейробиология пока и мечтать не осмеливается.

Для анализа Джонас и Кординг использовали классические методы, которыми нейробиология изучает настоящий человеческий мозг. Например, они специально

---

[1] *Митио Каку* (Michio Kaku, p. 1947) — американский ученый японского происхождения. Родители Каку познакомились в так называемом сегрегационном лагере «Тул Лейк» (Tule Lake), куда на время Второй мировой войны были депортированы многие жившие в США японцы. В школе любознательный Митио собрал в гараже работающий ускоритель, пытаясь получить антивещество. А сейчас заслуженный ученый Митио Каку известен широкой публике как популяризатор науки; мы искренне рекомендуем его основные книги, многие из которых переведены на русский язык: «Физика невозможного», «Физика будущего», особенно интересная в контексте этой книги «Будущее разума» и другие.

[2] Кстати говоря, название статьи – отсылка к уже ставшей классической статье Юрия Лазебника «Может ли биолог починить радиоприемник» [299], перепечатанной в начале 2000-х годов тремя разными журналами, включая Cell. В этой статье Лазебник пытается проанализировать сломанный радиоприемник «Океан» методами биологических наук со столь же неутешительными выводами.

симулировали повреждения отдельных транзисторов, чтобы узнать, за что они «отвечают». В рамках MOS 6502 они могли позволить себе попробовать повредить буквально *каждый* транзистор по отдельности. И действительно, они сумели выделить отдельные подмножества транзисторов, которые были необходимы для запуска каждой из трех рассмотренных игр (*Space Invaders*, *Donkey Kong* и *Pitfall*); без такого транзистора одна игра не запускалась, а две другие работали нормально. Можно было бы предположить, что эти транзисторы являются ключевыми для именно этого конкретного «поведения»... Вот только на самом деле не было ничего подобного: большинство этих транзисторов на самом деле реализовывали простые функции, например сложение, и чисто случайно именно эта часть реализации оказывалась нужна лишь в одной игре. Конечно, если бы исследователи заранее знали, что эта часть «мозга» реализует сложение, и затем начали повреждать отдельные транзисторы, смысла в этом было бы больше. Но ведь в реальной нейробиологии мы тоже совершенно не умеем изолировать настолько простые, базовые функции.

Джонас и Кординг применяют и другие методы, часто использующиеся в современной нейробиологии (они уже сложнее, и мы не будем здесь подробно их объяснять), со столь же переменным успехом: иногда получаются верные умозаключения, но не менее часто получаются неверные, и никакого способа отличить одни от других «нейробиология Atari» не дает.

Другой пример, показывающий огромную сложность изучения и моделирования настоящих мыслительных процессов — это то, как сложно ученым по-настоящему убедительно если не понять, то хотя бы просимулировать даже простейшие системы нейронов. Например, совсем недавно стало известно о важном успехе на этом пути: проект OpenWorm [404, 536] начал работу по симуляции нервной системы нематоды *C. elegans*. Этот червь — один из самых хорошо изученных современной биологией организмов. Он стал для современной генетики примерно тем же, чем муха дрозофила была для классической; о нем опубликованы тысячи работ, есть даже специальная конференция под названием *C. elegans* Genetics, и практически все механизмы его нехитрого нематодьего существования давно известны ученым. Проект OpenWorm поставил целью сделать достаточно достоверную модель всех 959 клеток *C. elegans*, а начать решили с того, чтобы полностью моделировать его нервную систему и то, как она приводит к движениям червя. Система эта состоит из целых 302 нейронов (одна из самых простых нервных систем в мире) и 95 клеток мышечного волокна. Проект, реализуемый согласно философии открытого программного обеспечения, движется успешно, но дело оказалось совсем не таким простым, как могло показаться: в 2015 году координатор проекта Стивен Ларсон признавал, что они пока «прошли только 20–30 % пути».

Мы не слишком сомневаемся в успехе проекта OpenWorm, пусть он и движется медленнее, чем хотелось бы его авторам. Его история достаточно наглядно показывает, что мы еще очень далеки от понимания более крупномасштабных нервных систем, не говоря уж о человеческом мозге. Но мечтать не запретишь: в 2013 году Евросоюз выделил 1,3 миллиарда евро проекту Генри Маркрама (Henry Markram) со

скромным названием *Human Brain Project* (HBP). В рамках проекта планировалось построить компьютерную модель всего человеческого мозга целиком, со всеми его миллиардами нейронов и триллионами синапсов. Маркрам, конечно же, не мошенник, обманом втершийся в доверие лиц, принимающих решения о будущем европейской науки, а блестящий нейробиолог. Моделировать мозг — его давняя мечта, проект всей его жизни; он уже руководил вполне успешным Blue Brain Project, построив компьютерную модель маленького кусочка мозга крысы (одной колонки неокортекса). Но все же цель HBP на данный момент, пожалуй, слишком амбициозна; неудивительно, что начавшийся с большой помпой проект уже через год подвергся суровой критике из-за плохого управления, а с 2015 года Маркрам был отстранен от руководства, и вся эта инициатива была серьезно реструктурирована.

Да и сами глубокие сети, о которых мы будем говорить в этой книге, не перестают удивлять исследователей. Хотя в данном случае люди сами запрограммировали этот «мыслительный аппарат» и твердо знают, как работает каждый конкретный «нейрон» сети, есть работы, посвященные буквально изучению свойств глубоких нейронных сетей, как будто они были бы природным объектом, данным нам для изучения (собственно, они таковым и являются, с той лишь разницей, что все же представляют собой математическую абстракцию). Например, в широко известной работе [249] рассказано, как можно обмануть сети, распознающие изображения (даже такие простые изображения, как черно-белые рукописные цифры), с помощью микроизменений, не влияющих на человеческое восприятие. Обученная глубокая сеть здесь выступает как «черный ящик», интересные свойства которого мы пытаемся понять. В этой книге мы увидим примеры такого подхода.

Значит ли это, что мы ничего не знаем о мозге и любые попытки провести аналогии между тем, что делают нейронные сети, и тем, как работает мозг, совершенно неуместны? На наш взгляд, вовсе нет. Конечно, наши знания о мозге не очень детальны, и результаты исследований того, как работают глубокие сети, никак нельзя напрямую переносить на функционирование человеческого мозга (да и крысиного лучше поостеречься) — слишком много там еще непонятного для нас. Да и ставить перед собой цель как можно точнее смоделировать человеческий мозг нейронной сетью — возможно, не лучшая идея. Однако все это совершенно не запрещает нам *вдохновляться* тем, как работает мозг, использовать архитектурные *идеи*, приходящие из нейробиологии, для того, чтобы улучшить работу нейронных сетей. В этой книге мы увидим несколько очень удачных параллелей между алгоритмами и архитектурами, использовавшимися в нейронных сетях, и представлениями современной нейробиологии о том, как устроен наш мозг. Просто надо понимать, что результатом здесь на данный момент могут быть только более удачные модели для решения конкретных прикладных задач. До задачи «повторить архитектуру человеческого мозга» мы, скорее всего, все-таки еще не доросли.

Однако многие другие задачи уже вполне доступны! В заключительном разделе этой главы мы дадим краткий обзор того, на что сейчас способны глубокие нейронные сети, и поговорим о задачах, которые для них пока недоступны.

1.6. Блеск и нищета современных нейронных сетей

> Даже если мы воздержимся от столь вселенского охвата, нам следует учесть все многообразие проявлений любви. Не только мужчина любит женщину, а женщина любит мужчину; мы любим также искусство и науку, мать любит своего ребенка, а верующий любит Бога.
>
> *Х. Ортега-и-Гассет. Этюды о любви*

> А теперь — слайды.
>
> *Анекдот*

Итак, мы увидели, какие задачи пытается решать машинное обучение. Вся эта книга будет посвящена тому, как нейронные сети с ними справляются. Но прежде чем углубляться в суть дела и начинать рассказывать о математике нейронных сетей, мы решили мотивировать читателей, да и себя, кратким обзором самых ярких *применений* глубоких нейронных сетей, а также рассказом об их ограничениях и дальнейших перспективах. Итак, для чего же используются нейронные сети сегодня, где уже достигнуты самые яркие успехи, а где еще остается много работы?

Первым значительным индустриальным приложением современных глубоких нейронных сетей, которое подтвердило, что революция глубокого обучения действительно начинает необратимо менять ландшафт машинного обучения, да и вообще мира вокруг нас, стали успехи в *распознавании речи*. Структура полноценного распознавателя речи выглядит так:

1) сначала звуковой сигнал преобразуется в признаки специального вида;
2) затем эти признаки превращаются в гипотезы, которые предлагают варианты конкретных фонем для каждого окна в звуковом сигнале;
3) потом гипотезы о фонемах объединяются в гипотезы относительно произнесенных слов, и в выборе между ними уже участвует не только обработка самого звука, но и языковые модели.

До глубоких сетей первые два шага этого процесса выглядели так: сначала звуковой сигнал превращался в так называемые MFCC-признаки[1], фонемы из них распознавали с помощью скрытых марковских моделей [119], а языковые модели представляли собой обычно сглаженные распределения n-грамм, то есть модели, которые оценивают вероятность следующего слова при условии нескольких предыдущих [281] (мы подробно поговорим о языковых моделях в разделе 7.2).

[1] MFCC расшифровывается как mel-frequency cepstral coefficients, то есть от сигнала нужно сначала взять преобразование Фурье, получив *спектр*, затем перейти к логарифмам амплитуд частот на шкале мелов, а потом взять от этих логарифмов обратное преобразование Фурье, получив *кепстр* (это слово получилось обращением первых четырех букв слова «спектр») [379].

Глубокие сети начали с того, что заменили собой распознаватель, основанный на скрытых марковских моделях. Более того, быстро стало ясно, что MFCC-признаки тоже можно улучшить путем обучения: нет, сети пока что не начинают работу непосредственно с необработанного звукового сигнала, но представления, подающиеся на вход современным системам распознавания, гораздо более «сырые», чем MFCC [260].

Первые такие исследования, основанные на глубоких сетях с предобучением без учителя, появились около 2010 года (см. обзоры [109, 116]), а уже к 2012-му это привело к тому, что все крупнейшие игроки на рынке распознавания речи перешли на нейронные сети: и Microsoft [481], и Google [7, 295], и IBM [346]. Основные результаты здесь были достигнуты сначала благодаря тому, что выделение признаков для декодеров, основанных на скрытых марковских моделях, можно было «отделить» от самих декодеров и отдать глубоким сетям, то есть сначала это было во многом обучение без учителя. Но вскоре появились и прорывные результаты о распознавании end-to-end, то есть полностью от начала до конца с учителем [96], и сейчас фактически все проекты для распознавания речи, включая виртуальных помощников вроде Google Now и Apple Siri, работают на глубоких нейронных сетях. Сюда же примыкает и анализ музыки: хотя там успехи пока не столь впечатляющие, последние модели для распознавания и порождения музыки тоже работают на глубоких нейронных сетях [55].

Вслед за речью пришло время обработки изображений. Точнее, она была всегда: с 1980-х годов в группе Яна ЛеКуна изображения обрабатывали именно нейронными сетями. К тому же времени относится и первый взлет *сверточных нейронных сетей*, специальной архитектуры, которая отлично подходит для обработки именно таких входов, как картинки; сверточным сетям мы посвятим главу 5 [18, 205]. В целом, это редкий пример области, в которой нейронные сети никогда полностью не пропадали из виду. Однако после начала революции глубокого обучения прогресс в обработке изображений тоже резко ускорился. В 2009–2010 годах глубокие сверточные сети выиграли ряд соревнований по распознаванию символов [396] и даже распознаванию видео с камер слежения [259, 430]. Кроме того, в 2009 году появились первые реализации нейронных сетей на графических процессорах [434], что дало огромный импульс всем связанным со сверточными сетями исследованиям (видеокарты для сверточных сетей особенно полезны).

И началось: в 2010 году были серьезно превзойдены давние результаты на классическом датасете распознавания рукописных цифр MNIST (мы расскажем о нем в разделе 3.6, и многие примеры будут с ним связаны), а еще через пару лет появились первые системы, которые распознавали изображения лучше, чем люди! В 2011-м глубокие сверточные сети лучше людей распознавали дорожные знаки на фотографиях [162, 216], что очень важно для автоматического вождения автомобилей, а с 2014 года Facebook распознает лица наших друзей не хуже, чем мы сами [112]. В наши дни самые глубокие сети — это именно сверточные сети для обработки изображений или видео; они могут насчитывать несколько сотен слоев.

Достижения глубоких нейронных сетей в *обучении с подкреплением* тоже трудно переоценить. Глубокие сети оказались как нельзя кстати, потому что дают универсальный «черный ящик», способный приблизить функцию «оценки позиции». Даже первые значительные успехи обучения с подкреплением в конце 1980-х уже были основаны на нейронных сетях. Последний громкий успех глубоких сетей в обучении с подкреплением — созданная DeepMind программа AlphaGo, которая сумела обыграть одного из лучших игроков мира в го — одну из последних классических игр с полной информацией, которые считались крайне сложными для компьютера[1]; об этом мы будем подробно говорить в главе 9.

Кроме игр, есть, конечно, и «более серьезные» приложения (в кавычках, потому что такие игры, как го или *StarCraft*, — это, как вы понимаете, на самом деле очень серьезно). Так, глубокое обучение уже находит применение и в другой традиционной области обучения с подкреплением — *роботике*. Во-первых, глубокие сети можно использовать для обработки сигналов, в частности визуальных, помогая роботу понять, что его окружает. Например, так глубокие сети применяются в *беспилотных автомобилях* (self-driving cars), которые сейчас уже вовсю разрабатывают не только Tesla и Google, но и другие крупнейшие автомобильные концерны, и даже Яндекс. Глубокие сети здесь можно использовать и напрямую. Например, в недавней яркой работе исследователей из NVIDIA глубокая сверточная сеть получала на вход картинку с установленной на автомобиле камеры, а на выход подавала уже непосредственные команды управления рулем; и вроде бы все получилось [142]. Во-вторых, глубокие сети можно использовать и в обучении с подкреплением для роботики [110, 158]; об этом мы поговорим в разделе 9.5.

Но в некоторых областях нейронным сетям все еще, мягко скажем, есть куда стремиться. Например, хотя большие успехи в области задач обработки естественного языка несомненны, и мы посвятим этим задачам отдельную главу 7, пока еще неясно, когда же наконец компьютер действительно научится читать, то есть обрабатывать содержащуюся в тексте информацию. Например, современные системы для ответов на вопросы пока что умеют понимать только очень простые истории и отвечать на вопросы вроде «Вася взял книгу; Вася пошел на кухню; Вася взял шоколадку и вернулся в комнату; где сейчас книга?». До человеческого уровня там еще очень, очень далеко.

---

[1] Кстати, вот яркий пример того, что такие обзоры писать одновременно и очень просто — применений у глубоких сетей пруд пруди, куда ни копнешь, будет интересно — и очень сложно, — как у Ахиллеса с черепахой: никогда не получается дописать до конца, все время появляется что-то новое. Вот и сейчас: мы собирались рассказать о том, что основная классическая игра, в которую компьютеры пока еще не могут превзойти человека, — это покер (no-limit Texas hold'em, если быть точным, потому что с лимитными играми уже некоторое время назад разобрались). Но в конце января 2017 года появилась новость о программе Libratus, которая очень уверенно обыграла команду из четырех человек, являющихся одними из лучших онлайн-профессионалов мира [63], а затем появилась и модель DeepStack, которая уже несомненно основана на глубоких нейронных сетях [113]. А исследователи из DeepMind в качестве следующей цели упоминали StarCraft — как ни странно, сейчас, в начале 2017 года, в компьютерную игру с немалой долей микроменеджмента люди пока что играют заметно лучше.

Чем же отличается задача понимания текста от задачи распознавания дорожных знаков? Чего не хватает современным нейронным сетям? Можно ли довести их до человеческого уровня понимания, и если да, то как?

В недавней работе [66] специалисты по когнитивным наукам (среди авторов, в частности, известные когнитивисты и специалисты по байесовскому выводу Джошуа Тенненбаум и Самуэль Гершман) попытались дать ответ на эти вопросы. Они начинают с того, что выделяют разницу между тем, как обучается человек, и тем, как это делают программы даже в успешных случаях. Человеку практически всегда нужно гораздо меньше тренировочных данных, чтобы успешно обучиться: например, чтобы освоить одну из игр Atari, глубокой сети требуется около тысячи часов опыта; это стало очень ярким шагом на пути развития глубокого обучения с подкреплением [238]. А человеку, даже ранее абсолютно незнакомому с игрой, для повторения этих результатов достаточно пару минут посмотреть YouTube, чтобы понять, что надо делать, и потом еще полчаса поиграть самому, чтобы воплотить понимание в навык. А в задаче распознавания символов (MNIST, стандартный датасет для распознавания цифр, мы уже упоминали, и он будет центральным примером в этой книге) человеку обычно достаточно буквально одного-двух изображений, чтобы начать узнавать новый символ даже в других формах и с серьезными искажениями. Почему же люди настолько лучше обучаются?

Во-первых, у человека еще в раннем детстве появляется понимание нескольких крайне важных для нормального функционирования основных предметных областей — то, что называется *базовым знанием* (core knowledge) [503]. Кроме понимания операций с числами и множествами и навигации в пространстве, в [66] особенно подчеркиваются два компонента, которые отличают живого человека от искусственных нейронных сетей.

1. *Интуитивная физика*: понимание того, как работает окружающий нас физический мир. Младенцы довольно быстро начинают разбираться в окружающем их физическом мире. К шести месяцам ребенок уже хорошо понимает постоянство объектов физического мира, то, что они должны двигаться непрерывно и не менять мгновенно свою форму, и уже даже различают твердые и жидкие объекты [502]. А к году малыши уверенно овладевают такими понятиями, как инерция, поддержка, способность одних объектов содержаться в других и т. д. [23, 375]. У ученых нет уверенности в том, как это все работает у человека, но по современным представлениям, интуитивная физика — это нечто вроде логических рассуждений, построенных на модели физической симуляции, вроде физической модели в компьютерной игре [239]. Модель эта, конечно, крайне приблизительная, мы не проводим настоящих вычислений, но достаточно точная для повседневных выводов и, главное, способная к очень мощным обобщениям и переносу на новые визуальные входы. Но пока неясно, как передать это понимание обучающейся модели. Недавняя работа исследователей из Facebook AI Research [320] начинает применять нейронные сети для развития подобной интуиции, но здесь, пожалуй, еще довольно далеко даже до первых неуверенных шагов младенца.

2. *Интуитивная психология*: тут дела обстоят еще интереснее. Дети, даже младенцы, очень быстро понимают, что некоторые сущности в окружающем мире обладают волей и действуют, преследуя какие-то свои цели. Более того, уже годовалые малыши прекрасно могут различать эти цели и действия по их достижению [403] и даже делать некие зачатки моральных суждений, понимая, когда «плохие» агенты мешают «хорошим» достигать их «хороших» целей [203]. Именно такие рассуждения позволяют нам с вами быстро научиться компьютерной игре, просто наблюдая за тем, что делают другие люди, и осознавая, какие цели они преследуют. Вы можете даже ни разу не увидеть, как Марио умирает, столкнувшись с черепашкой, — просто посмотрев на то, что опытный игрок все время избегает или убивает черепашек, вы поймете, что это враг и с ним нужно поступать именно так. Откуда все это берется — пока тоже до конца не понятно; возможно, из развивающейся сейчас в когнитивистике байесовской теории сознания [25] (о байесовском подходе к обучению мы начнем говорить в разделе 2.1, а современным нейробайесовским исследованиям посвятим главу 10), а возможно, и нет. Но понятно, что такого рода рассуждения искусственные нейронные сети сейчас совершенно не умеют вести, и это может быть важным направлением для дальнейших исследований.

Во-вторых, люди очень хороши в том, что называется *переносом обучения* (transfer learning): мы можем быстро построить модель нового объекта или процесса, порождая правильные абстракции из очень, очень маленького числа обучающих примеров. Известно, что дети с трех до 14–15 лет изучают в среднем 8–9 новых слов каждый день[1]. Довольно очевидно, что они не могут получить большое число разных контекстов для каждого нового слова и должны обучаться по считанным единицам тренировочных примеров. Сейчас начинают проводиться исследования о том, как перенести такое обучение (его обычно называют обучением по одному примеру, one-shot learning) в нейронные сети и модели машинного обучения в целом. С помощью байесовского подхода к обучению уже достигнуты некоторые успехи в таких задачах, как распознавание рукописных символов [290] и речевых сигналов [402], но основная работа здесь еще впереди. Например, в разделе 9.4 мы поговорим о программе AlphaGo, которая недавно победила человека-чемпиона в игре го. Сможет ли AlphaGo разумно сыграть с человеком, например, в го на гексагональной или тороидальной доске? Вряд ли. А человек, умеющий играть в го, адаптируется мгновенно, и хотя, конечно, не сразу достигнет тех же высот в новом варианте, сразу начнет играть вполне разумно.

В-третьих, настоящим камнем преткновения для искусственного интеллекта остается *причинность* (causality), то есть способность распознавать и выделять «истинные причины» наблюдаемых эффектов, строить модели процессов, которые могли бы привести к таким наблюдениям. Когда человек смотрит на фотографию, в его воображении обычно создается некий нарратив, объясняющий происходящее

[1] Есть ссылки на интересные исследования того, как именно дети обучаются новым словам [51], но на самом деле это довольно очевидное рассуждение: просто разделите словарный запас взрослого человека (около 30 тысяч слов) на те 10 лет, в течение которых его основная масса нарабатывается.

на снимке как связную последовательную историю. А когда нейронная сеть порождает подписи к фотографиям, ничего подобного не происходит; часто бывает так, что сеть корректно распознает все ключевые объекты на фото, но не может связать их правильным логическим образом [274]. Это, разумеется, обычно связано и с вышеупомянутыми интуитивной физикой и интуитивной психологией: они помогают нам выбрать правильное объяснение. Психологические и когнитивные исследования показывают, что причинность в этом смысле может появляться и на более низком уровне: например, классические теории восприятия речи утверждают, что ее проще всего объяснить через «обращение» услышанного, распознавание движений речевого тракта, которые могли бы привести к таким звукам [418].

И наконец, люди гораздо лучше *умеют учиться*. Младенцы учатся относительно медленно, но затем способность к обучению постепенно «раскручивается», и мы с вами осваиваем новое гораздо более эффективно, чем имеющиеся у нас данные позволили бы, например, современным архитектурам нейронных сетей. Это показывает, что у людей при обучении появляются очень сильные ограничения, априорные распределения. Первые шаги в направлении такого «абстрагирования» уже, конечно, делаются. Например, довольно очевидно, что даже самые разнообразные системы компьютерного зрения могут переиспользовать первые уровни анализа изображения, выделение базовых признаков, которые могут оставаться общими для самых разных изображений (мы поговорим об этих уровнях и вообще о современных системах компьютерного зрения в главе 5). Однако это пока все еще только первые шаги, причем ограниченные конкретными областями применения — здесь мы еще только в начале большого пути.

Что ж, современные нейронные сети действительно еще очень далеки от «настоящего» искусственного интеллекта. Но есть у специалистов по глубоким сетям и позитивная программа о том, как двигаться вперед. Так, например, в работе [361] изобретатель самого популярного метода распределенных представлений слов *word2vec* (о нем мы подробно поговорим в главе 7) Томаш Миколов и его соавторы попытались изложить свое видение того, как нейронные сети могут двигаться в сторону AI. По мнению Миколова, есть два основных качества «настоящего» интеллекта, которые нужны, чтобы признать программу разумной:

- *способность к коммуникации*, чтобы программа могла интерактивно общаться с людьми и получать информацию об окружающем мире; для этого в [361] предлагается использовать единый канал связи общего вида, в который и обучающийся агент, и те, кто его обучают, могут писать сообщения разнообразной формы;
- *способность к обучению*, и в части собственно обучения, и в части *мотивации* для этого обучения, которая должна приходить через тот самый канал для коммуникации с окружающим миром в форме положительных и отрицательных стимулов (как в обучении с подкреплением).

Для этого в работе [361] предлагается построить специальную экосистему для обучающихся агентов, похожую на компьютерную игру. В этой экосистеме будет

Учитель (Teacher), чье поведение полностью контролируется экспериментаторами, и Ученик (Learner), которого мы, собственно, и выращиваем. Они общаются через вышеозначенный канал связи, по которому также отдельно передаются стимулы (награды) за желаемое и нежелательное поведение Ученика. Идея системы состоит в том, чтобы Учитель последовательно обучал Ученика задачам возрастающей сложности:

- сначала просто обращать внимание на то, что говорит Учитель, и понимать, что эти команды и ответы как-то связаны с наградами[1];
- затем подавать команды правильной формы для окружающей среды, исследовать ее и двигаться в ней (эту среду можно представлять себе как компьютерный текстовый квест из конца восьмидесятых: Ученик пишет команды вроде move left, open door или look around, а среда сообщает результаты и выдает награды по необходимости);
- обобщать отдельные команды, замечать в них закономерности; например, выделить класс «объектов», над которыми можно проводить одинаковые манипуляции, или класс «команд поворота» в разные стороны;
- затем научиться следовать командам более высокого уровня, например «пройди вперед два раза» или «найди яблоко»;
- перейти к интерактивному диалогу с Учителем, когда у Учителя можно выяснять какую-то информацию, необходимую для выполнения команды (например, где находится то самое яблоко);
- и наконец, добраться до реализации настоящих алгоритмов, то есть научиться подавать в окружающую среду команды с циклами и условиями, для которых окружающая среда может выступать как простой компьютер.

Переходить между этими «уровнями» можно за счет изменения структуры вознаграждений: когда Учитель видит, что Ученик обучился давать простые команды, он перестает поощрять за любые корректные команды и начинает вознаграждать только за приводящие к нужному эффекту; затем «нужные эффекты» усложняются и т. д. Миколов с соавторами пишут, что основным компонентом систем, которые могли бы обучиться абстрактному мышлению, должна стать долгосрочная память в той или иной форме; она должна быть способна хранить обученные моделью паттерны поведения, присваивать им ярлыки (например, запомнить, что такое «найти яблоко») и затем «доставать» нужные паттерны по этим ярлыкам. Существующие сейчас нейронные сети вряд ли способны к такому уровню абстракции при интерактивном обучении; нужны новые идеи.

Мы так подробно описали эту систему не потому, что верим, будто бы именно она непременно приведет к созданию разумных компьютеров в ближайшем будущем. Скорее наоборот, мы привели этот пример для того, чтобы стало понятно, что

[1] Обратите внимание: в этой системе предполагается, что награды определены отдельно, и Ученику не нужно понимать, что положительная награда — это хорошо. В целом, проблема мотивации и целеполагания пока остается больным вопросом искусственного интеллекта.

хотя современные модели машинного обучения и делают множество потрясающих вещей, постепенно превосходя человека во многих ранее недостижимых для компьютера областях, до «универсального черного ящика», который мог бы самостоятельно обучиться действовать в новой обстановке, как это делает человек, пока еще очень, очень далеко. Так что нам придется закончить эту вводную главу не закрытой, а открытой каденцией: несмотря на все громкие успехи, у машинного обучения все еще впереди, и вполне возможно, что от вас, дорогие читатели, зависит, как быстро мы сможем от смелых мечтаний о текстовых квестах дойти до настоящего искусственного интеллекта. Дерзайте!

Глава 2
Предварительные сведения,
или Курс молодого бойца

TL;DR

Во второй главе мы дадим краткий обзор предварительных сведений, требующихся для того, чтобы дальше перейти непосредственно к нейронным сетям. А именно, мы рассмотрим:

- основы теории вероятностей, теорему Байеса и вероятностный подход к машинному обучению;
- функции ошибки в машинном обучении и регуляризацию;
- различие между регрессией и классификацией, функции ошибки для классификации;
- главный метод оптимизации в нейронных сетях — градиентный спуск;
- конструкцию графа вычислений и алгоритмы дифференцирования на нем;
- и наконец, практическое введение в библиотеку TensorFlow, которую мы будем использовать для примеров в этой книге.

2.1. Теорема Байеса

> В работе предложен новый подход к актуальной проблеме — духовно-нравственному воспитанию молодежи. В ходе исследования устанавливается, что практически все основоположники теории вероятностей имели прямое или косвенное отношение к Святой Церкви. Случайно или закономерно такое совпадение?
>
> *С. Н. Дворяткина. Роль математики случайного в духовно-нравственном воспитании молодежи: поиск истины, Вестник МГОУ, Серия «Педагогика», № 4, 2009, с. 79–84*

> Вера и вероятность — очень близки друг другу не только филологически.
>
> *К. А. Чхеидзе, из письма Н. В. Устрялову*

В этой главе мы поговорим о нескольких основополагающих для машинного обучения вещах, приведем необходимые предварительные сведения о тех разделах математики, которые понадобятся для понимания дальнейшего, а также дадим практическое введение в библиотеку TensorFlow, которая будет использоваться в большинстве примеров книги. Эта глава фактически никак напрямую не связана с нейронными сетями, и искушенный читатель ее вполне может пропустить безо всякого ущерба для понимания. Однако возьмем на себя риск порекомендовать читателю чуть менее опытному все же пробежать эту главу глазами — возможно, вы найдете здесь что-то новое.

Первый сюжет, важнейший не только для обучения глубоких нейронных сетей, но и для всего машинного обучения, — это *вероятностная* интерпретация машинного обучения и вообще *байесовский* взгляд на окружающий наш мир. Как мы уже видели в предыдущей главе, машинное обучение — это наука о том, как на основании данных делать выводы, откуда эти данные взялись, и предсказания, какие данные встретятся нам в будущем. Важно, что делать точные выводы невозможно: процессы, приводящие к порождению данных, слишком сложны даже в самых простых случаях[1]. Наши модели всегда, неизбежно будут содержать некоторую долю *неопределенности*; а математическое описание неопределенности и операций с неопределенными величинами дает как раз теория вероятностей. Более того, вероятностный подход к обучению зачастую позволяет нам не только делать предсказания, но и оценивать, насколько мы уверены в этих предсказаниях.

[1] Простейший пример: если знать заранее все параметры броска монеты: начальную скорость, состояние окружающего воздуха, распределение массы и т. п., то теоретически вполне возможно рассчитать ее полет и достаточно уверенно предсказать, чем она выпадет. Но это настолько сложно, что мы не задумываясь используем монету как честный генератор случайных чисел.

Поэтому машинное обучение как наука основано на теории вероятностей. Кстати говоря, именно в обучении нейронных сетей это не всегда просто заметить. Ниже мы увидим, что нейронные сети можно интерпретировать так: мы вводим некие довольно логичные целевые функции, а потом их оптимизируем; вот и все, при чем тут вероятности. Тем не менее мы скоро увидим, что теория вероятностей все равно является тем фундаментом, на котором основано все происходящее, и устойчивое владение основами этой науки совершенно необходимо для понимания и данной книги, и других книг о машинном обучении.

На этом месте читатель, воспитанный классическими университетскими курсами теории вероятностей [595–597], наверняка недовольно поморщился: в голове его промелькнули полузабытые определения борелевских подмножеств пространства $\mathbb{R}^n$, алгебры и сигма-алгебры, вероятности как меры на сигма-алгебре борелевских подмножеств и т. д., и т. п. И действительно, когда на теорию вероятностей начали смотреть как на теорию меры, был сделан один из важнейших шагов к формализации первой: аксиоматика Колмогорова[1] поразительно похожа на аксиомы теории меры, и этот единый взгляд позволил сильно развить теорию вероятностей как науку.

Однако наша задача проще. Для того чтобы читать эту книгу и вообще для того чтобы понимать почти все происходящее в современном машинном обучении (кроме, возможно, некоторых пока что довольно эзотерических областей байесовского вывода), вполне достаточно не вспомнить целиком университетский курс теории вероятностей, а воспользоваться тем, что обычно в реальности остается в голове после того, как вы этот курс прослушали. Для дальнейшего нам вполне достаточно понимать, что:

- бывают дискретные случайные величины с конечным или счетным набором исходов; каждому из своих исходов они присваивают неотрицательную вероятность, и вероятности исходов в сумме дают единицу; классический и фактически единственный пример здесь — бросание кубика;

[1] *Колмогоров, Андрей Николаевич* (1903–1987) — советский математик, один из величайших математиков XX века. В юности Колмогоров занимался историей, а позже любил рассказывать, как ушел из нее: оказалось, что в истории каждый вывод должен быть подкреплен несколькими доказательствами, и Колмогоров «решил уйти в науку, где одного доказательства было бы достаточно». Математическую карьеру он начал с математического анализа, быстро перешел к основаниям математики, доказав важный результат об интуиционистской логике, а затем перешел к теории вероятностей. Колмогоров — буквально создатель всей современной теории вероятностей; он первым сформулировал аксиоматику теории вероятностей, основанную на теории меры, и доказал ряд основополагающих результатов. И все это было лишь первым взлетом творчества Колмогорова; следующий пришелся на 1950-е годы, когда Андрей Николаевич получил выдающиеся результаты о динамических системах, небесной механике и представления функций. В информатике Колмогоров предложил новое понятие алгоритма и разработал теорию так называемой колмогоровской сложности... нет, поля этой книги слишком малы для всех его достижений. Кстати, в искусственный интеллект Колмогоров верил; в статье [590] он писал, что «принципиальная возможность создания полноценных живых существ, построенных полностью на дискретных (цифровых) механизмах переработки информации и управления, не противоречит принципам материалистической диалектики».

- бывают одномерные непрерывные случайные величины, у которых набор исходов представляет собой вещественную прямую $\mathbb{R}$; тогда вероятности отдельных исходов превращаются в *функцию распределения* $F(a) = p(x < a)$ и ее производную (в этом месте может быть много сложностей, но практически всегда в наших примерах функция F будет непрерывно дифференцируемой), *плотность распределения*:

$$p(x) = \frac{\mathrm{d}F}{\mathrm{d}x};$$

теперь не сумма, а интеграл неотрицательной функции плотности должен быть равен единице:

$$\int_{-\infty}^{\infty} p(x)\mathrm{d}x = F(\infty) - F(-\infty) = 1.$$

Все эти определения по сравнению с «настоящей» теорией вероятностей сильно упрощены, да и вообще не очень формальны. Но для нужд машинного обучения их нам будет вполне достаточно.

Идем дальше. *Совместная вероятность* — это вероятность одновременного наступления двух событий, $p(x, y)$. Грубо говоря, если есть два кубика, на каждом из которых могут выпасть числа от 1 до 6, то мы можем рассмотреть новую случайную величину «два кубика», у которой будут возможные исходы (1,1), (1,2), (1,3) и так далее до (6, 6), всего $6 \times 6 = 36$ исходов. Две случайные величины называются *независимыми*, если:

$$p(x, y) = p(x)p(y).$$

Обратите внимание, что независимость в теории вероятностей определяется сугубо формально. Ее не надо путать ни с отношением «причина — следствие» (каузальностью), которого между зависимыми случайными величинами может и не быть, ни с корреляцией, которая отражает только *линейную* часть зависимости между случайными величинами.

Чтобы получить обратно из совместной вероятности вероятность того или иного исхода одной из случайных величин, нужно просуммировать по другой:

$$p(x) = \sum_y p(x, y).$$

Этот процесс иногда называется умным словом *маргинализация*: если рассмотреть ее в случае непрерывных случайных величин, получится, что мы фактически проецируем двумерное распределение, поверхность в трехмерном пространстве, на одну из осей, получая функцию от одной переменной:

$$p(x) = \int_Y p(x, y)\mathrm{d}y.$$

Условная вероятность — вероятность наступления одного события, если известно, что произошло другое, $p(x \mid y)$; ее обычно определяют формально так:

$$p(x \mid y) = \frac{p(x, y)}{p(y)}.$$

Аналогично обычной независимости можно теперь определить *условную независимость*: x и y условно независимы при условии z, если

$$p(x, y \mid z) = p(x \mid z)p(y \mid z).$$

Чтобы проиллюстрировать все эти базовые определения, в качестве примера рассмотрим известный «парадокс» Монти Холла (в кавычках потому, что никакого парадокса здесь на самом деле нет). Представьте себе, что на игровом шоу усатый ведущий предлагает вам выбрать из трех абсолютно одинаковых шкатулок: в одной из них лежат деньги, в двух других ничего нет. Вы принимаете решение (пока что у вас нет никаких оснований предпочесть одну из шкатулок), после чего ведущий открывает одну из двух оставшихся и показывает, что в ней пусто. Очевидно, он всегда может так сделать: даже если вы выбрали пустую шкатулку, из двух оставшихся все равно одна пустая найдется. И теперь наступает момент истины: ведущий предлагает вам изменить свое решение и взять вместо выбранной ту шкатулку, которая осталась закрытой. Выгодно ли вам это делать?

Многие предполагают, что нет никакой разницы. И действительно, рассмотрим события наличия денег в трех шкатулках, обозначив их через x_1, x_2 и x_3 соответственно. Изначально их вероятности были равны: $p(x_1) = p(x_2) = p(x_3) = \frac{1}{3}$. Если бы ведущий просто открыл одну из шкатулок (для определенности пусть это будет x_3) до вашего выбора, то две другие шкатулки остались бы равновероятными: $p(x_1 \mid x_3 = 0) = p(x_2 \mid x_3 = 0) = \frac{1}{2}$.

Но структура эксперимента устроена сложнее! Ведущий открывает не случайную пустую шкатулку, а одну из двух не выбранных игроком. Поэтому события «какую из двух пустых шкатулок открыть» и «лежат ли деньги в выбранной вами шкатулке» становятся зависимыми. Давайте без потери общности предположим, что вы изначально выбрали первую шкатулку, а выбранную ведущим шкатулку обозначим через y. Будем предполагать, что если деньги действительно лежат в выбранной вами первой шкатулке ($x_1 = 1$), то ведущий выбирает одну из двух пустых равновероятно. Тогда совместные вероятности разложатся так:

$$p(x_1 = 1, y = 2) = \frac{1}{6}, \qquad p(x_2 = 1, y = 2) = 0, \qquad p(x_3 = 1, y = 2) = \frac{1}{3},$$
$$p(x_1 = 1, y = 3) = \frac{1}{6}, \qquad p(x_2 = 1, y = 3) = \frac{1}{3}, \qquad p(x_3 = 1, y = 3) = 0,$$

и это совсем не похоже на определение независимых величин, правда?

Вернемся к парадоксу Монти Холла чуть позже, а сейчас продолжим разговор об основах теории вероятностей. По определению условной вероятности:

$$p(x, y) = p(x \mid y)p(y) = p(y \mid x)p(x),$$

и теперь можно выразить, например:

$$p(y \mid x) = \frac{p(x \mid y)p(y)}{p(x)} = \frac{p(x \mid y)p(y)}{\sum_{y' \in Y} p(x \mid y')p(y')}.$$

Стоп. Последняя формула, которая у нас получилась, — это, конечно, всего лишь очень простое формальное следствие определения условной вероятности. Но вместе с тем это самая главная формула всего машинного обучения, — *формула*, или *теорема Байеса*[1]. В ней действительно нет ничего сложного, однако именно выводы, основанные на теореме Байеса, становятся ключевыми и в машинном обучении, и просто в наших житейских рассуждениях. Дело в том, что формула Байеса позволяет переоценивать наши априорные представления о мире (в формуле выше это $p(y)$) на основе частичной информации (данных), которую мы получили в виде наблюдений (в формуле выше это $p(x \mid y)$), в качестве вывода получая новое состояние наших представлений $p(y \mid x)$.

О такой интерпретации мы еще поговорим ниже, а на этом месте, пожалуй, не удержимся от того, чтобы привести пример из классической области применения статистики — медицины. Его постоянно приводят в книгах и лекциях о байесовском выводе и машинном обучении, но от долгого употребления он не только не истерся, а скорее даже засиял новыми красками.

Предположим, что некий тест на какую-нибудь страшную болезнь имеет вероятность успеха 95 %; иначе говоря, 5 % — это вероятность как ошибки первого рода (ложного срабатывания, false positive), так и ошибки второго рода (пропуска больного человека, false negative). Предположим также, что болезнь очень распространена и имеется у 1 % респондентов. Отложим на время то, что респонденты могут быть разного возраста и профессий — будем предполагать, что больные люди в нашем эксперименте выбираются из популяции случайно и равномерно.

Пусть теперь некий человек (все так же случайно и равномерно выбранный из популяции) получил позитивный результат теста, то есть тест говорит, что страшная болезнь у человека присутствует. С какой вероятностью он действительно болен? Попробуйте, прежде чем подсчитать или прочитать ответ, оценить свою

[1] *Томас Байес* (Thomas Bayes, 1702–1761) — английский пресвитерианский священник, богослов и математик. При жизни Байес опубликовал только две работы: «Благость Господня, или Попытка Доказать, что Конечной Целью Божественного Провидения и Направления Является Счастье его Созданий» и «Введение в Доктрину Флюксий и Защита Математиков от Возражений Автора “Аналитика”». Вторая работа была опубликована анонимно; в ней Байес защищал математический анализ от критики Джорджа Беркли. Ключевая для нас работа Байеса была опубликована только посмертно, по представлению Ричарда Прайса. Однако современники хорошо знали Байеса как математика: несмотря на отсутствие официальных работ, в 1742 году его избрали в члены Лондонского Королевского общества.

интуицию: как бы вы оценили вероятность болезни, если бы получили позитивный результат от такого теста?

Давайте сначала подсчитаем результат точно. Обозначим через t результат теста, через d — наличие болезни. Тогда:

$$p(t=1) = p(t=1 \mid d=1)p(d=1) + p(t=1 \mid d=0)p(d=0).$$

Используем теорему Байеса:

$$p(d=1 \mid t=1) = \frac{p(t=1 \mid d=1)p(d=1)}{p(t=1 \mid d=1)p(d=1) + p(t=1 \mid d=0)p(d=0)} = \\ = \frac{0{,}95 \times 0{,}01}{0{,}95 \times 0{,}01 + 0{,}05 \times 0{,}99} = 0{,}16.$$

Иначе говоря, получилось, что вероятность действительно оказаться больным — всего 16 %! Почему так мало? На самом деле, если вдуматься в условия задачи, ответ 16 % покажется достаточно ясным: грубо говоря, из 100 % у вас всего 1 % на то, чтобы оказаться действительно больным, и 5 % на то, что тест ошибся и выдал неверный положительный результат. Значит, условная вероятность быть больным при условии положительного теста примерно равна $\frac{1}{1+5} = \frac{1}{6}$. Это грубый подсчет, совсем не учитывающий ошибку теста в другую сторону, но порядок величины получается верный.

Такие задачи составляют суть вероятностного вывода (probabilistic inference). Поскольку они обычно основаны на теореме Байеса, вывод часто называют байесовским (Bayesian inference). Но не только поэтому; есть и еще одна важная причина. Дело в том, что в классической теории вероятностей, происходящей из физики, вероятность обычно понимается как предел отношения количества определенного результата эксперимента к общему количеству экспериментов. Стандартный пример здесь — это бросание монеты: если бросить честную монету тысячу раз, число выпавших решек будет довольно близко к пятистам, хотя вряд ли точно равно 500 (вот, кстати, небольшое упражнение на понимание: какова будет точная вероятность получить ровно 500 решек из 1000 бросаний честной монеты?).

Задача о медицинском тестировании, которую мы сейчас решали, является примером так называемой *обратной задачи* теории вероятностей. *Прямые* задачи теории вероятностей возникают, когда дано описание некоего вероятностного процесса или модели, а найти требуется вероятность того или иного события, то есть фактически по модели предсказать поведение. Например: в урне лежат десять шаров, из них три черных; какова вероятность выбрать черный шар? Или: в урне лежат десять шаров с номерами от 1 до 10; какова вероятность того, что номера трех последовательно выбранных шаров дадут в сумме 12?

А обратные задачи, напротив, просят по известному поведению некоего стохастического объекта построить вероятностную модель. Например: перед нами две

урны, в каждой по десять шаров, но известно, что в одной три черных, а в другой — шесть. Кто-то взял из одной из урн шар, и шар оказался черным. Насколько вероятно, что шар брали из первой урны?

Такую же обратную задачу можно решить и в парадоксе Монти Холла. Раз величины y (какую шкатулку откроет ведущий) и x_1 (будут ли деньги в выбранной шкатулке) оказались зависимыми, можно предположить, что тот факт, что мы узнали y, может помочь нам по-новому оценить вероятность x_1. И действительно:

$$p(x_1 = 1 \mid y = 2) = \frac{p(x_1 = 1, y = 2)}{p(x_1 = 1, y = 2) + p(x_1 = 0, y = 2)} =$$

$$= \frac{p(x_1 = 1, y = 2)}{p(x_1 = 1, y = 2) + p(x_2 = 1, y = 2) + p(x_3 = 1, y = 2)} = \frac{1/6}{1/6 + 1/3} = \frac{1}{3},$$

$$p(x_1 = 0 \mid y = 2) = \frac{p(x_1 = 0, y = 2)}{p(x_1 = 1, y = 2) + p(x_1 = 0, y = 2)} =$$

$$= \frac{p(x_2 = 1, y = 2) + p(x_3 = 1, y = 2)}{p(x_1 = 1, y = 2) + p(x_2 = 1, y = 2) + p(x_3 = 1, y = 2)} = \frac{1/3}{1/6 + 1/3} = \frac{2}{3}.$$

Это значит, что в парадоксе Монти Холла действительно выгодно изменить свое решение. Ответ, опять же, вполне интуитивен, если посмотреть под правильным углом: вы фактически выбираете между одной шкатулкой и *обеими* оставшимися, потому что из тех двух выбор уже сделали за вас[1].

Классическая, «фриквентистская» (то есть основанная на частотах) теория вероятностей рассуждает о том, как оперировать вероятностями повторяющихся событий, когда можно устремить число экспериментов к бесконечности; например, высказывание «монета выпадает решкой с вероятностью $\frac{1}{2}$» означает, что при большом числе подбрасываний решек получится примерно половина, причем чем больше экспериментов, тем ближе отношение должно быть к $\frac{1}{2}$. Но в жизни часто возникает необходимость оценить вероятность событий, к которым такие рассуждения совершенно неприменимы. Например, насколько вероятно, что:

- сборная России победит на ближайшем чемпионате мира по футболу;
- компания Google обанкротится в течение ближайших 20 лет;

[1] Но любопытно, что, когда этот кажущийся парадокс был представлен широкой публике в журнале Parade [554], в колонке Мэрилин вос Савант, считающейся человеком с самым высоким IQ в мире, редакция получила буквально тысячи писем! В этих письмах люди, часто с учеными степенями, возмущенно объясняли, что задача вводит читателей в заблуждение, и правильный ответ — одна вторая, ведь «очевидно», что шкатулки остались независимыми. Парадокс Монти Холла до сих пор считается одним из самых ярких примеров того, что люди плохо приспособлены для интуитивных рассуждений с вероятностями; см., например, книгу [189], посвященную связанным с парадоксом Монти Холла психологическим исследованиям.

- «Одиссею» написала женщина (некоторые исследователи выдвигают такую версию, и звучит она, кстати, довольно правдоподобно);
- все мы живем внутри компьютерной симуляции (это тоже интересное рассуждение: если компьютерные симуляции целых вселенных вообще возможны, то вполне вероятно, что мы в одной из них и находимся);
- мы выпустим второе издание этой книги?

Все это события, о вероятностях которых вполне хочется рассуждать, а иногда и нужно рассуждать для решения практических задач; например, шансы сборной России пока еще, кажется, интересуют букмекеров. Но о «стремящемся к бесконечности числе экспериментов» здесь говорить бессмысленно — эксперимент ровно один. Можно, конечно, всласть натеоретизироваться о «возможных мирах» и квантовой неопределенности, но эти рассуждения все равно не помогут нам провести много экспериментов и узнать их результаты.

В таких случаях вероятности уже выступают как *степени доверия* (degrees of belief). В такой интерпретации можно рассматривать теорию вероятностей как некое расширение обычной пропозициональной логики. В классической логике речь идет о строгих законах вывода, верных всегда, но можно рассмотреть и правила вывода для операций, работающих с вероятностями различных высказываний. Оказывается, что эти правила вывода будут подозрительно напоминать аксиоматику «обычной» теории вероятностей.

Например, если я считаю, что с вероятностью $0{,}2$ завтра пойдет дождь, а с вероятностью $0{,}7$ мне нужно будет завтра идти на работу, и делаю предположение о том, что события эти независимы, значит, с вероятностью $0{,}7 \cdot 0{,}2 = 0{,}14$ я завтра пойду на работу под дождем. Число $0{,}14$ здесь представляет собой или попытку объективной оценки правдоподобия того события, что я завтра пойду под дождем на работу (обратите внимание, что событие по-прежнему уникально, повторяющихся экспериментов с погодой не бывает), — такой взгляд называют *объективистским*, — или, в *субъективистском* подходе, просто представляет собой мое личное мнение об этом будущем событии. Заметьте, что в обратных задачах вероятности сразу стали байесовскими: в задаче о медицинском тестировании мне интересно, насколько правдоподобно то, что болен лично я со своим уникальным положительным результатом теста (в данном примере, впрочем, задачу легко переформулировать через частоты, но это не всегда так).

Все это и составляет *байесовский подход* к вероятностям. Сам термин появился в середине XX века, сначала в книге Гарольда Джеффриса (Harold Jeffreys) «Теория вероятностей» [261], а затем в работах первых «байесианистов» Абрахама Вальда (Abraham Wald) [556] и Леонарда Сэвиджа (Leonard Savage) [470]. Ну а сама идея применения теоремы Байеса для пересчета вероятностей между априорными и апостериорными гипотезами, как ни странно, действительно восходит к работе Томаса Байеса «Очерки к решению проблемы доктрины шансов» (An Essay towards solving a Problem in the Doctrine of Chances), вышедшей уже после смерти автора, в 1763 году [36].

В работе Байеса впервые вводилось достаточно строгое определение понятия условной вероятности и решалась типичная задача байесовского вывода, обратная задача теории вероятностей: предположим, что некто пронаблюдал, как из урны с лотерейными билетами достали десять пустых билетов и один выигрышный. Какова будет вероятность того, что отношение между пустыми и выигрышными билетами в урне находится между 9:1 и 11:1? Байес доказал, что для десяти билетов она будет невелика, около 7,7 %, а затем установил эти вероятности для большего числа наблюдений (вплоть до 1000 выигрышей из 10 000 билетов, где эта вероятность достигает 97 %), а также вывел общую формулу.

К сожалению, сейчас мы уже не можем установить, был ли сам преподобный Томас Байес байесианистом, то есть понимал ли он рассчитываемые им вероятности как степени доверия или как соотношения между результатами многочисленных экспериментов[1]. К счастью, и те, и другие вероятности подчиняются одинаковым законам; классические результаты Ричарда Кокса показывают, что вполне естественные аксиомы вероятностной логики тут же приводят к весьма узкому классу функций, фактически полностью определяя теорию вероятностей [99].

Сейчас байесовский подход стал общепринятым: нет сомнений, что неопределенность формализовывать нужно в рамках теории вероятностей, а пересчитывать вероятности при получении новой информации — по теореме Байеса. Но как все это связано с машинным обучением?

Оказывается, теорема Байеса — это основной, центральный инструмент машинного обучения, на ней держатся буквально все рассуждения этой книги и многие другие. Чтобы это увидеть, давайте запишем все ту же теорему Байеса в немного иных обозначениях:

$$p(\boldsymbol{\theta} \mid D) = \frac{p(\boldsymbol{\theta})p(D \mid \boldsymbol{\theta})}{p(D)} = \frac{p(\boldsymbol{\theta})p(D \mid \boldsymbol{\theta})}{\int_{\boldsymbol{\theta}\in\Theta} p(D \mid \boldsymbol{\theta})p(\boldsymbol{\theta})\mathrm{d}\boldsymbol{\theta}}.$$

И введем общепринятую в машинном обучении терминологию:

- $p(\boldsymbol{\theta})$ — *априорная вероятность* (prior probability);
- $p(D \mid \boldsymbol{\theta})$ — *правдоподобие* (likelihood);
- $p(\boldsymbol{\theta} \mid D)$ — *апостериорная вероятность* (posterior probability);
- $p(D) = \int p(D \mid \boldsymbol{\theta})p(\boldsymbol{\theta})\mathrm{d}\boldsymbol{\theta}$ — *вероятность данных* (evidence).

[1] Любопытно, однако, как Байеса интерпретировал известный философ-моралист, проповедник и математик Ричард Прайс, который представил работу Байеса и выступил ее формальным соавтором при посмертной публикации. Прайс считал, что теорема Байеса дает нам еще одно (телеологическое) доказательство бытия Бога: «Цель здесь состоит в том, чтобы показать, почему мы считаем, что в общем положении вещей есть фиксированные законы, по которым происходят события, и тем подтвердить аргумент в пользу существования Бога от конечных целей... Легко показать, что обратная задача, решаемая в этой работе, напрямую применима для данной цели: она показывает нам точно и ясно, в каждом случае всякого возможного порядка произошедших событий, на каких основаниях мы можем полагать, что этот порядок был произведен из неизменных причин или правил, установленных природой, а не из беспорядочных случайностей».

Практически все задачи машинного обучения имеют вид некоторой модели с параметрами $\boldsymbol{\theta}$; задача состоит в том, чтобы по данным D подобрать описывающие их параметры $\boldsymbol{\theta}$ наилучшим образом[1]. В классической статистике для этого обычно ищут *гипотезу максимального правдоподобия* (maximum likelihood, ML):

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} p(D \mid \boldsymbol{\theta}),$$

где $\arg\max_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$ — это значение вектора $\boldsymbol{\theta}$, на котором достигается максимум функции $f(\boldsymbol{\theta})$. А в байесовском подходе и современном машинном обучении ищут *апостериорное распределение* (posterior):

$$p(\boldsymbol{\theta} \mid D) \propto p(D \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}),$$

а затем, если нужно, *максимальную апостериорную гипотезу* (maximum a posteriori hypothesis, MAP):

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid D) = \arg\max_{\boldsymbol{\theta}} p(D \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}).$$

Значок $\propto$ здесь означает «пропорциональность»: на самом деле $p(\boldsymbol{\theta} \mid D)$ не равно $p(D \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})$, а только пропорционально этому произведению; чтобы получить распределение вероятностей, нужно еще нормализовать результат. Но поскольку нормировочная константа $\int p(D \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta}$ не зависит от $\boldsymbol{\theta}$, ее часто можно не учитывать. Например, при максимизации $\arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid D) = \arg\max_{\boldsymbol{\theta}} p(D \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})$. Мы часто будем пользоваться этим обозначением.

Разницу легко увидеть на примере простой задачи вывода: предположим, что нам дали потенциально нечестную монетку, мы ее несколько раз подбросили, и теперь знаем последовательность результатов бросков. Давайте попробуем определить ее «нечестность» и предсказать, что выпадет в следующий раз. Правдоподобие заданной последовательности результатов бросков монеты, среди которых всего h решек и t орлов, составляет

$$p(h, t \mid \theta) = \theta^h (1 - \theta)^t,$$

где параметр θ означает вероятность выпадения решки. Максимизировать эту функцию по θ несложно: гипотеза максимального правдоподобия скажет, что вероятность решки равна числу выпавших решек, деленному на число экспериментов.

Иначе говоря, это значит, что если вы взяли незнакомую монетку, подбросили ее один раз и она выпала решкой, вы теперь ожидаете, что она всегда будет выпадать только решкой, правильно?

---

[1] Иногда в машинном обучении говорят о непараметрических методах; обычно в таких случаях имеется в виду не то, что у них нет параметров, а то, что число параметров у них заранее неизвестно и тоже в некотором смысле является параметром.

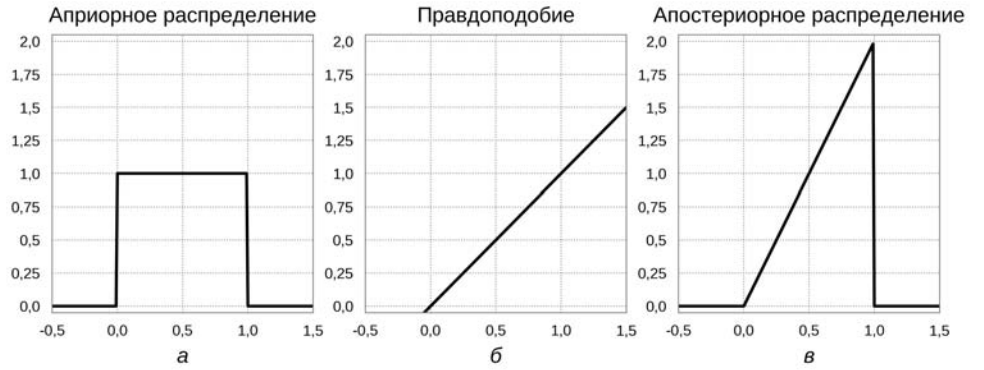


Рис. 2.1. Один бросок незнакомой монеты:
а — априорное распределение $p(\theta) = 1, \theta \in [0; 1]$; *б* — правдоподобие $L(\theta) = \theta$;
в — апостериорное распределение $p(\theta) \propto \theta, \theta \in [0,1]$

Странно получается... На самом деле ведь мы вовсе не придем к такому выводу, а если бы пришли, он бы почти наверняка не подтвердился. Кстати, и формально этот результат сомнителен: правдоподобие равно $L(\theta) = \theta$, и максимум не ограничен, если устремить $\theta \to \infty$. Конечно, есть «интуитивно понятное» ограничение: вероятность выпадения решки не может быть меньше 0 и не может быть больше 1. Но как его формализовать?

В реальности мы, даже исследуя незнакомую монетку, уже заранее чего-то ожидаем от нее; формализацией наших ожиданий и является *априорное распределение* $p(\boldsymbol{\theta})$. Оно умножается на функцию правдоподобия и сглаживает ее. В наших рассуждениях априорное распределение всегда незримо присутствует, даже если явно оно не задано, так что лучше уж определить его явно. Даже от совершенно незнакомого процесса мы ожидаем, например, что для него все варианты вероятностей θ одинаково правдоподобны: возможно, монетка всегда выпадает орлом, возможно, всегда решкой, как знать?..

Но даже в этом случае у нас есть априорное распределение, просто оно равномерное на отрезке [0,1]: тот факт, что у нас нет предпочтений на θ, тоже можно считать определенного вида предпочтением. Это изображено на рис. 2.1: априорное распределение $p(\theta) = 1$ для $\theta \in [0,1]$ (и ноль вне этого отрезка) умножается на правдоподобие $L(\theta) = \theta$, и получается апостериорное распределение $p(\theta) \propto \theta$ для $\theta \in [0,1]$ и 0 для других θ.

Более того, если это монетка, которую мы только что достали из кармана, у нас достаточно сильна уверенность в том, что она близка к честной и априорное распределение $p(\theta)$ представляет собой достаточно острый «колокол» с максимумом в $\frac{1}{2}$. Один возможный пример такого «колокола» показан на рис. 2.2. В качестве априорного распределения мы выбрали *бета-распределение*

$$\text{Beta}(\theta; \alpha, \beta) = \frac{1}{B(\alpha,\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}, \text{ где } B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)},$$

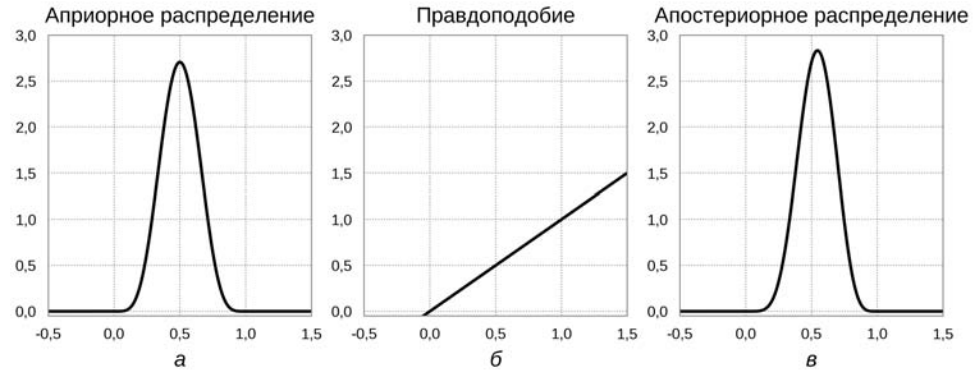


Рис. 2.2. Один бросок монетки с априорным распределением Beta(6,6): *а* — априорное распределение $p(\theta) = \text{Beta}(\theta; 6{,}6)$; *б* — правдоподобие $L(\theta) = \theta$; *в* — апостериорное распределение $p(\theta) = \text{Beta}(\theta; 6{,}7) \propto \theta\text{Beta}(\theta; 6{,}6)$

где Γ — гамма-функция, для натуральных чисел соответствующая факториалу: $\Gamma(a) = (a-1)!$. Выбор был, конечно, осознанный: бета-распределение очень похоже на функцию правдоподобия монетки $p(h,t \mid \theta) = \theta^h (1-\theta)^t$. И если умножить одно на другое и нормировать, получится опять бета-распределение:

$$\text{Beta}(\theta; \alpha, \beta) \times p(h, t \mid \theta) \propto \theta^{\alpha-1}(1-\theta)^{\beta-1}\theta^h (1-\theta)^t = \\ = \theta^{\alpha+h-1} (1-\theta)^{\beta+t-1} \propto \text{Beta}(\theta; \alpha+h, \beta+t).$$

Такие априорные распределения называются *сопряженными*. Если мы задаем сопряженное априорное распределение, байесовский вывод сводится к тому, чтобы каким-нибудь нехитрым, заранее раз и навсегда посчитанным способом подправить параметры этого априорного распределения. Например, в случае с монеткой получилось, что для перехода от априорного распределения к апостериорному достаточно просто прибавить число решек к первому параметру бета-распределения, а число орлов — ко второму. Это проиллюстрировано на рис. 2.2, где мы сделали переход от $\text{Beta}(\theta; 6{,}6)$ к $\text{Beta}(\theta; 6{,}7)$, что эквивалентно умножению на θ с последующей нормализацией.

Когда мы проводим байесовский вывод в задачах машинного обучения, кроме правдоподобия мы еще должны выбрать априорное распределение на всех возможных значениях параметров. А затем уже мы считаем и максимизируем $p(\boldsymbol{\theta} \mid D) \propto$ $\propto p(D \mid \boldsymbol{\theta})p(\boldsymbol{\theta})$, то есть ищем максимальную апостериорную гипотезу. Это вполне естественно: нас интересует именно распределение параметров при условии данных $p(\boldsymbol{\theta} \mid D)$, а не наоборот. И теорема Байеса позволяет нам перейти от $p(\boldsymbol{\theta} \mid D)$, которое зачастую совершенно непонятно как подсчитать, к вычислению отдельно правдоподобия $p(D \mid \boldsymbol{\theta})$, которое обычно и определяется в модели, и априорного

распределения $p(\boldsymbol{\theta})$, которое мы придумываем сами. Причем придумывать стараемся в такой форме, чтобы вычисление $p(\boldsymbol{\theta} \mid D)$ было как можно проще и точнее, желательно в виде аналитической формулы, как было показано выше. Поэтому теорема Байеса лежит в основе практически всех методов машинного обучения.

Можно сделать и следующий шаг: зачем вообще нам сдались эти параметры? Зачем нужно обучать апостериорное распределение на $\boldsymbol{\theta}$? Все наше моделирование требуется для того, чтобы научиться *предсказывать* ответы на последующие подобные вопросы: например, мы обучаемся отделять кошек от собак в имеющейся базе фотографий, чтобы затем отличить кошку от собаки на новом, ранее не виденном снимке. Поэтому на самом деле нас часто интересует даже не апостериорное распределение $p(\boldsymbol{\theta} \mid D)$, а *предсказательное распределение* (predictive distribution) $p(y \mid D)$ или $p(y \mid D, \boldsymbol{x})$, где y — это следующий пример в данных или правильный ответ на новый вопрос $\boldsymbol{x}$. Чтобы найти это распределение, нужно свести его к $p(y \mid \boldsymbol{\theta})$, которое определяется моделью, и апостериорному распределению $p(\boldsymbol{\theta} \mid D)$:

$$p(y \mid D) = \int_{\Theta} p(y \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid D) \mathrm{d}\boldsymbol{\theta} \propto \int_{\Theta} p(y \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) p(D \mid \boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta},$$

где через Θ мы обозначили множество, которому должны принадлежать различные значения вектора параметров $\boldsymbol{\theta}$.

Например, мы можем решить задачи байесовского вывода для нашего примера с нечестной монеткой, изображенного на рис. 2.1. Максимальная апостериорная гипотеза здесь не будет отличаться от гипотезы максимального правдоподобия, попрежнему получится $\theta_{\text{MAP}} = 1$. А вот байесовское предсказание уже даст нетривиальный результат:

$$p(h \mid D) = \int_{-\infty}^{\infty} p(h \mid \theta) p(\theta \mid D) \mathrm{d}\theta = \\ = \int_{-\infty}^{\infty} \theta p(\theta \mid D) \mathrm{d}\theta = \int_0^1 \frac{\theta^2}{\int_0^1 \theta' \mathrm{d}\theta'} \mathrm{d}\theta = \frac{1/3}{1/2} = \frac{2}{3}.$$

Мы получили частный случай так называемого *правила Лапласа*: как правильно сглаживать предсказания результата бинарного события по данным.

Однако найти предсказательное распределение — непростая задача даже для обычных линейных моделей, таких как линейная и логистическая регрессия. А в более сложных моделях она практически всегда требует достаточно сложных (как идейно, так и вычислительно) методов приближенного байесовского вывода. Впрочем, далее предсказательных распределений в нашей книге практически не будет. Найти полное апостериорное распределение на всех весах большой нейронной сети и проинтегрировать по нему все-таки совсем уж сложно, нам бы с апостериорным распределением разобраться.

А что все это значит в реальности, алгоритмически? Что мы будем делать, решая задачи машинного обучения? Будем решать задачи оптимизации. У нас получилось, что обучение практически любой модели машинного обучения сводится к задаче оптимизации либо апостериорного распределения:

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid D) = \arg\max_{\boldsymbol{\theta}} p(D \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})$$

(заметьте, что значок пропорциональности $\propto$ после взятия arg max превращается просто в равенство: знаменатель формулы Байеса в данном случае от $\boldsymbol{\theta}$ не зависит, и при оптимизации его учитывать не обязательно), либо на худой конец просто правдоподобия:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} p(D \mid \boldsymbol{\theta}).$$

Обычно мы будем предполагать, что каждая точка данных была порождена описанным в модели процессом независимо, то есть правдоподобие набора данных D будет представлять собой большое произведение по всем его точкам:

$$p(D \mid \boldsymbol{\theta}) = \prod_{d \in D} p(d \mid \boldsymbol{\theta}).$$

Поэтому часто удобно перейти к логарифмам и вместо произведения максимизировать сумму (логарифм — функция монотонная, и arg max опять не меняется):

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(D \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) \prod_{d \in D} p(d \mid \boldsymbol{\theta}) =$$

$$= \arg\max_{\boldsymbol{\theta}} \left(\log p(\boldsymbol{\theta}) + \sum_{d \in D} \log p(d \mid \boldsymbol{\theta}) \right).$$

Искусственную нейронную сеть тоже можно представить как один из примеров таких моделей; сейчас пока не будем делать этого формально — в конце концов, этому будет посвящена вся книга, — но при разговоре о регуляризации в нейронных сетях в главе 4 нам точно еще пригодится вероятностный взгляд на вещи.

Мы еще вернемся к байесовскому выводу и гораздо подробнее поговорим о современных вероятностных моделях в главе 10. Но там скорее нейронные сети будут помогать вести вывод в вероятностных моделях, а не наоборот. Для изучения же самих нейронных сетей нам будет вполне достаточно того необходимого минимума, который мы здесь изложили.

Подведем краткий итог, который нужно держать в голове при чтении этой книги, да и любого другого текста о машинном обучении:

- математическая модель в машинном обучении обычно представляет собой задание распределения вероятностей на данных и параметрах $p(\boldsymbol{\theta}, D)$;

- иногда совместное распределение параметров и данных $p(\boldsymbol{\theta}, D)$ моделируют напрямую, но чаще — в виде произведения правдоподобия $\boldsymbol{p}(D \mid \boldsymbol{\theta})$ и априорного распределения $p(\boldsymbol{\theta})$;
- апостериорное распределение $p(\boldsymbol{\theta} \mid D)$ по теореме Байеса можно получить как (нормированное) произведение $p(\boldsymbol{\theta})p(D \mid \boldsymbol{\theta})$;
- задача машинного обучения обычно состоит в том, чтобы найти и/или максимизировать распределение $p(\boldsymbol{\theta} \mid D)$, — важно понять, какие параметры лучше всего подходят к имеющимся данным и нашим априорным представлениям, а затем, при необходимости, делать новые предсказания из предсказательного распределения $p(x \mid D) = \int_{\Theta} p(x \mid \boldsymbol{\theta})p(\boldsymbol{\theta} \mid D)\mathrm{d}\boldsymbol{\theta}$;
- в реальности же все это обычно превращается в задачу оптимизации, обычно оптимизации логарифма $\log p(D \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$, который состоит из собственно логарифма правдоподобия модели и регуляризаторов.

О возникающих здесь задачах оптимизации мы сейчас вкратце и поговорим.

2.2. Функции ошибки и регуляризация

Когда не делаешь ошибок, перестаешь совершенствоваться.

Дж. Мартин. Пир для воронов

Какие из этих признаков составляют то, что признано называть «ошибкою», — это покамест еще не решено, но, во всяком случае, сомнение уже позволительно.

М. Е. Салтыков-Щедрин.
Ошибки молодости. Комедия Петра Штеллера

Мы узнали, что большинство задач машинного обучения сводятся к решению той или иной задачи оптимизации. Подробный разговор о том, как именно проводить эту оптимизацию, мы будем вести в главе 4, но сейчас нам потребуется немного забежать вперед и рассказать об основной идее происходящего. Часто бывает, что изложение в книге лучше всего строить по спирали. Сейчас мы очень кратко введем основной метод оптимизации — метод *градиентного спуска*, потом посмотрим на то, какие, собственно, функции мы хотели бы оптимизировать, а через несколько глав опять вернемся к самим алгоритмам. Тогда уже и обсудим подробно всевозможные модификации и улучшения метода градиентного спуска, которые обычно применяются в обучении современных нейронных сетей. При этом получится, что в начале главы 4 мы отчасти повторим то, что вы прочитаете здесь, но это представляется нам меньшим злом.

Итак, о градиентном спуске. На нем будут основываться почти все остальные методы оптимизации, которые мы будем рассматривать. Суть его работы легче всего проиллюстрировать, если представить себе трехмерную поверхность функции от двух аргументов; интуитивно эта поверхность является значением функции

ошибки от весов модели, которые мы обучаем[1]. Теперь представим, что текущее значение параметров — это небольшой шарик, находящийся на данной поверхности. Наша задача — найти минимум функции ошибки, и эта задача довольно точно соответствует тому, что шарик хотел бы сделать под действием силы тяжести, — скатиться в самую глубокую яму.

Сейчас наша задача — найти направление, в котором шарик будет катиться в такой ситуации. С реальным шариком все достаточно просто: понятно, что (при нулевой начальной скорости) он будет катиться в том направлении, в котором у поверхности самый большой наклон вниз. А если говорить формально — в направлении, прямо противоположном *градиенту* к поверхности. Давайте кратко напомним, что это такое. Если функцию, определяющую ту поверхность, на которой лежит шарик, обозначить через $E(\boldsymbol{\theta}) = E(\theta_1, \theta_2, \ldots, \theta_n)$, то ее градиент ∇E — это (вспоминаем курсы математического анализа) вектор производных функции нескольких переменных по каждой из компонент:

$$\nabla_{\boldsymbol{\theta}} E = \begin{pmatrix} \frac{\partial E}{\partial \theta_1} \\ \vdots \\ \frac{\partial E}{\partial \theta_{n-1}} \\ \frac{\partial E}{\partial \theta_n} \end{pmatrix}.$$

Иначе говоря, градиент — это то направление, в котором функция быстрее всего возрастает. А значит, направление, в котором она быстрее всего убывает, — это и есть направление, обратное градиенту, то есть $-\nabla_{\boldsymbol{\theta}} E$.

Именно в этом заключается интуиция, стоящая за методом градиентного спуска. Нужно только внести поправку на то, что у нас нет возможности точно промоделировать непрерывный процесс катящегося вниз шарика, поэтому мы дискретизируем время и продвигаемся шаг за шагом. Обозначая через $\boldsymbol{\theta}_t$ вектор параметров модели на шаге t, а через E — минимизируемую функцию, мы можем записать вектор обновления параметров на шаге t так:

$$\boldsymbol{u}_t = -\eta \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}_{t-1}), \qquad \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \boldsymbol{u}_t.$$

Давайте даже на будущее запишем это в виде псевдокода:

```
u = - learning_rate * grad
theta += u
```

Здесь, конечно, сразу возникает много вопросов. Первый — как именно подсчитать градиент в каждой точке для нейронной сети со множеством весов, сложно

[1] Полезный совет от Джеффри Хинтона, одного из главных специалистов по нейронным сетям в мире: когда вы научились визуализировать график функции от двух аргументов, то есть трехмерную поверхность, это интуитивное понимание легко продолжить до n-мерных поверхностей. Для этого достаточно представить себе трехмерную поверхность и четко произнести про себя: «эн».

связанных друг с другом; пока будем предполагать, что делать это мы умеем и градиент известен на каждом шаге. Второй — как выбирать скорость обучения и нужно ли менять ее со временем (конечно, нужно!); об этом мы тоже поговорим позже, в главе 4. Там же и рассмотрим разные модификации, которые позволят существенно улучшить и ускорить градиентный спуск.

А сейчас давайте рассмотрим вопрос куда более основополагающий: что за функцию E мы собрались оптимизировать? Откуда она возьмется?

В предыдущем разделе мы говорили о том, что оптимизируется обычно апостериорная вероятность:

$$p(\boldsymbol{\theta}) \prod_{d \in D} p(d \mid \boldsymbol{\theta}),$$

или (чаще) ее логарифм:

$$\log p(\boldsymbol{\theta}) + \sum_{d \in D} \log p(d \mid \boldsymbol{\theta}).$$

Что это за функция в реальных задачах? Какие обычно делаются вероятностные предположения и к каким задачам оптимизации они приводят?

Начнем с самого простого случая: классической задачи *линейной регрессии*. Мы будем строить линейную модель имеющихся данных. Рассмотрим такую функцию:

$$y(\boldsymbol{x}, \boldsymbol{w}) = w_0 + \sum_{j=1}^{p} x_j w_j = \boldsymbol{x}^\top \boldsymbol{w}$$

для вектора входов $\boldsymbol{x} = (1, x_1, \ldots, x_p)$. Обратите внимание, что здесь мы внесли свободный член в вектор весов, добавив еще одну фиктивную размерность, вход по которой всегда равен единице, — это просто переобозначение, которое часто делают для удобства и сокращения записи. Таким образом, по вектору входов $\boldsymbol{x}^\top = (x_1, \ldots, x_p)$ мы будем предсказывать выход y так:

$$\hat{y}(\boldsymbol{x}) = \hat{w}_0 + \sum_{j=1}^{p} x_j \hat{w}_j = \boldsymbol{x}^\top \hat{\boldsymbol{w}}.$$

Давайте сначала попробуем рассуждать безо всякой теории вероятностей. Как найти оптимальные параметры $\boldsymbol{w}^*$ по тренировочным данным $D = \{ (\boldsymbol{x}_i, y_i)\}_{i=1}^N$? Для этого логично выбрать какую-нибудь функцию ошибки. Часто используют так называемый метод наименьших квадратов, в котором минимизируют сумму квадратов отклонений предсказанных значений от истинных:

$$\mathrm{RSS}(\boldsymbol{w}) = \sum_{i=1}^{N} (y_i - \boldsymbol{x}_i^\top \boldsymbol{w})^2.$$

В данном конкретном случае, кстати, задачу минимизации $\mathrm{RSS}(\boldsymbol{w})$ можно решить точно — функция ошибки оказывается выпуклой, более того, квадратичной, и чтобы найти ее минимум, достаточно решить систему линейных уравнений. Запишем функцию ошибки:

$$\mathrm{RSS}(\boldsymbol{w}) = (\boldsymbol{y} - \mathcal{X}\boldsymbol{w})^\top (\boldsymbol{y} - \mathcal{X}\boldsymbol{w}),$$

где $\mathcal{X}$ — матрица размера $N \times p$.

Продифференцируем по $\boldsymbol{w}$, и в предположении, что матрица $\mathcal{X}^\top\mathcal{X}$ невырожденная, получится:

$$\boldsymbol{w}^* = (\mathcal{X}^\top\mathcal{X})^{-1}\mathcal{X}^\top\boldsymbol{y}.$$

Можно найти $\boldsymbol{w}^*$ и градиентным спуском, для выпуклых функций он тоже всегда прекрасно работает. Но откуда вообще взялась эта странная идея? Почему мы минимизировали именно сумму квадратов отклонений, а не сумму модулей, четвертых степеней или экспонент? Неужели просто потому, что так было удобнее искать минимум?

Чтобы ответить на эти вопросы, давайте поговорим о линейной регрессии по-байесовски. Для этого нужно ввести вероятностные предположения. И основное предположение в модели линейной регрессии состоит в том, что шум (ошибка в данных) распределен *нормально* с центром в нуле, то есть переменная t, которую мы наблюдаем, получается так:

$$t = y(\boldsymbol{x}, \boldsymbol{w}) + \epsilon, \quad \text{где} \quad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

Иными словами:

$$p(t \mid \boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \mathcal{N}(t \mid y(\boldsymbol{x}, \boldsymbol{w}), \sigma^2).$$

Рассмотрим теперь такой же набор данных, как выше, $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ со значениями $\boldsymbol{t} = \{t_1, \ldots, t_N\}$. Будем предполагать, что эти значения были получены независимо, и шум всегда имел одно и то же распределение:

$$p(\boldsymbol{t} \mid \mathcal{X}, \boldsymbol{w}, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}\left(t_n \mid \boldsymbol{w}^\top \boldsymbol{x}_n, \sigma^2\right).$$

Вспомним, как выглядит плотность нормального распределения. Для одномерной случайной величины плотность такова:

$$p(x \mid \mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Подставим ее в формулу для правдоподобия и прологарифмируем:

$$\ln p(\boldsymbol{t} \mid \boldsymbol{w}, \sigma^2) = -\frac{N}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{n=1}^{N}\left(t_n - \boldsymbol{w}^\top \boldsymbol{x}_n\right)^2.$$

И вот мы получили, что для максимизации правдоподобия по $\boldsymbol{w}$ нам нужно как раз минимизировать среднеквадратичную ошибку!

Пока мы не двинулись дальше, отметим один важный момент. На первый взгляд может показаться, что нет никакой разницы между тем, чтобы просто выбрать функцию ошибки или распределение шума: между этими двумя объектами действительно есть соответствие.

Однако обратите внимание, как вероятностный (пока что даже не байесовский) подход к линейной регрессии сразу же вынудил нас явно выписать все сделанные предположения! Теперь мы знаем, что сумма квадратов отклонений соответствует нормально распределенному шуму с нулевым средним[1] — и если вдруг увидим, что ошибки у нас очевидно имеют другую природу (например, если бывают погрешности только «в плюс», а «в минус» не бывают), сможем догадаться поискать другую функцию ошибки.

К сожалению, некорректное применение статистических методов встречается очень часто. Авторам этой книги не раз приходилось видеть даже ученых, применявших статистические методы, предполагающие нормально распределенный шум, к дискретным данным, имеющим всего два возможных значения! Очевидно, получались не слишком осмысленные результаты. Таких проблем можно было бы избежать, если бы процесс применения этих методов заставлял явно выписать вероятностные предположения.

Но хватит морализаторства, наше дело еще не окончено. Следующий пункт плана — *регуляризация*. Когда параметров у модели становится очень много, она начинает слишком хорошо «облизывать» точки из тренировочного набора данных, а ее предсказательная способность от этого страдает. Это легко увидеть на еще одном избитом классическом примере. Давайте рассмотрим ту же самую линейную регрессию, но теперь добавим в нее базисные функции и будем искать не прямую, а многочлен, наилучшим образом описывающий данные точки. Иначе говоря, давайте считать, что входная переменная x всего одна и мы ищем оптимальные коэффициенты многочлена степени d, описывающего данные из $D = \{\,(x_i, y_i)\}_{i=1}^{N}$, то есть моделируем выходную переменную так:

$$\hat{y}(x) = w_0 + \sum_{j=1}^{d} w_j x^j = \begin{pmatrix}1 & x & x^2 & \dots & x^d\end{pmatrix}^\top \boldsymbol{w}.$$

[1] Вопрос для самопроверки читателя: а почему вообще естественно рассматривать нормально распределенный шум? Почему нормальное распределение так часто возникает в реальной жизни как распределение шумов и ошибок в непрерывных измерениях? Подсказка: ответ связан с законом больших чисел и центральной предельной теоремой.

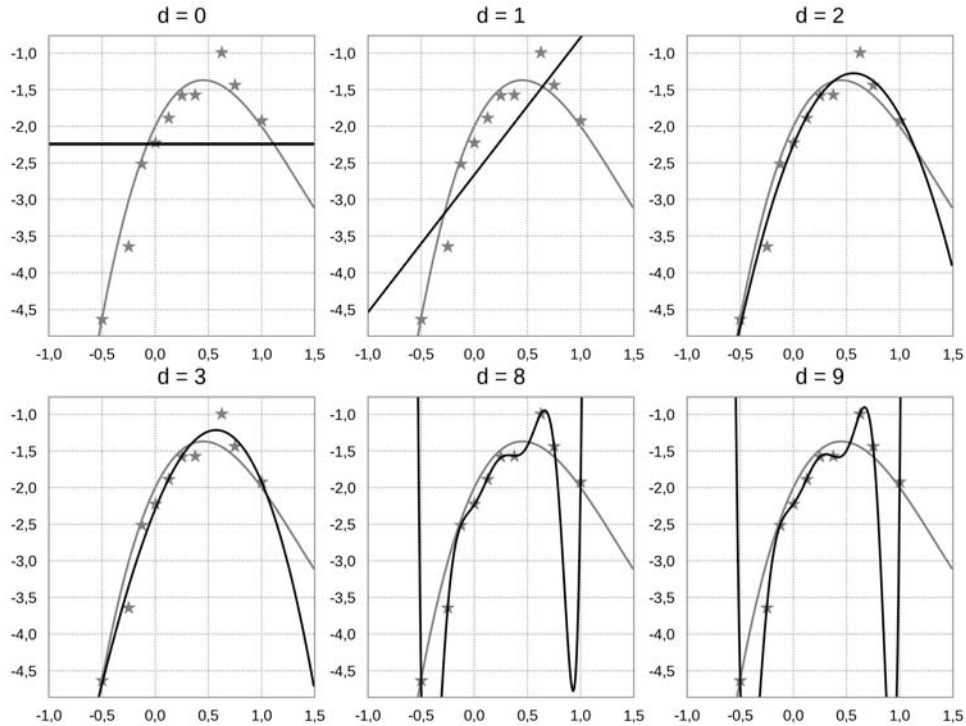


Рис. 2.3. Аппроксимация многочлена $f(x) = x^3 - 4x^2 + 3x - 2$ по десяти точкам с нормально распределенным шумом многочленами разной степени

Давайте посмотрим, что получается на конкретном примере. В качестве исходных точек мы взяли значения многочлена $f(x) = x^3 - 4x^2 + 3x - 2$ в восьми точках и добавили к результатам нормально распределенный шум с нулевым средним и дисперсией $\frac{1}{4}$.

Результаты изображены на рис. 2.3: на графиках исходный многочлен показан серой кривой, точки данных — звездочками; черные кривые — это результат аппроксимации. Видно, что для $d = 0$ мы получаем просто среднее всех точек, для $d = 1$ — оптимальную прямую. Для $d = 2$ и $d = 3$ результат получается весьма неплохой (что логично, ведь исходный многочлен был кубический), но для $d = 8$ «хвосты» многочлена уже смотрят в неправильные стороны (то есть интерполяция более или менее получилась, а экстраполяция — совсем нет), а для $d = 9$ полиномиальное приближение полностью вырождается. Черная кривая теперь приближает исходные точки *абсолютно точно*, ведь через десять точек можно точно провести многочлен девятой степени, но вот результаты этой аппроксимации теперь практически никакого отношения к исходному многочлену не имеют.

Обнаружилась проблема — чем больше степень многочлена, тем, конечно, точнее им можно приблизить данные, но в какой-то момент результаты приближения перестанут иметь отношение к действительности. Это классический пример *оверфиттинга*[1]. Как нам с ним справиться? Давайте сначала подойдем с несколько неожиданной стороны. В некоторый момент коэффициенты многочлена начинают очень сильно расти. Вот как выглядят изображенные на рис. 2.3 многочлены:

$$
\begin{aligned}
f_0(x) &= -\,2{,}2393,\\
f_1(x) &= -\,2{,}6617 + 1{,}8775x,\\
f_2(x) &= -\,2{,}2528 + 3{,}4604x - 3{,}0603x^2,\\
f_3(x) &= -\,2{,}2937 + 3{,}5898x - 2{,}6538x^2 - 0{,}5639x^3,\\
f_8(x) &= -\,2{,}2324 + 2{,}2326x + 6{,}2543x^2 + 15{,}5996x^3 - 239{,}9751x^4 +\\
&\quad + 322{,}8516x^5 + 621{,}0952x^6 - 1478{,}6505x^7 + 750{,}9032x^8,\\
f_9(x) &= -\,2{,}22 + 2{,}01x + 4{,}88x^2 + 31{,}13x^3 - 230{,}31x^4 +\\
&\quad + 103{,}72x^5 + 869{,}22x^6 - 966{,}67x^7 - 319{,}31x^8 + 505{,}64x^9.
\end{aligned}
$$

Коэффициенты получаются гораздо больше, чем мы могли бы предположить о нашей кривой априори (опять это слово...). Давайте попробуем найти способ с этим справиться. Сначала будем действовать прямолинейно и простодушно: добавим размер коэффициентов в функцию ошибки в качестве дополнительного слагаемого; такие слагаемые называются *регуляризаторами*. Раньше мы минимизировали такую функцию ошибки:

$$\text{RSS}\left(L(\boldsymbol{w})\right) = \frac{1}{2}\sum_{i=1}^{N}(f(x_i, \boldsymbol{w}) - y_i)^2,$$

а теперь будем минимизировать другую:

$$\text{RSS}(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{N}(f(x_i, \boldsymbol{w}) - y_i)^2 + \frac{\lambda}{2}\left\|\boldsymbol{w}\right\|^2,$$

где λ — коэффициент регуляризации, который нам самим нужно будет как-то выбрать. Оптимизировать эту функцию ошибки можно точно так же — и градиентный спуск сработает, да и по сути это по-прежнему квадратичная функция:

[1] Часто оверфиттинг (от одноименного английского overfitting) по-русски называют «переобучением». Хотя русское слово, безусловно, приятнее уху, нам все же кажется, что смысл у слова «переобучение» уже есть: «обучение заново, обучение чему-то другому», то есть не *over*-learning (что это вообще значило бы для человека?), а скорее *re*-learning. Поэтому будем пользоваться некрасивой калькой.

$$\mathrm{RSS}(\boldsymbol{w}) = \frac{1}{2}\left(\boldsymbol{y} - \mathcal{X}\boldsymbol{w}\right)^{\top}\left(\boldsymbol{y} - \mathcal{X}\boldsymbol{w}\right) + \frac{\lambda}{2}\boldsymbol{w}^{\top}\boldsymbol{w}.$$

От нее можно взять производную, приравнять нулю и решить уравнение:

$$\boldsymbol{w}^{*} = (\mathcal{X}^{\top}\mathcal{X} + \lambda\mathbb{I})^{-1}\mathcal{X}^{\top}\boldsymbol{y},$$

где $\mathbb{I}$ обозначает единичную матрицу соответствующей размерности. Метод регуляризации, при котором мы добавляем к функции ошибки $\frac{\lambda}{2}\|\boldsymbol{w}\|^2$, называется *гребневой регрессией* (ridge regression).

Позволим себе небольшое лирическое отступление. Само слово «регуляризация» здесь выглядит немного странно; это потому, что оно имеет скорее исторический смысл. Дело в том, что изначально речь шла о решении линейных уравнений вида $A\boldsymbol{x} = \boldsymbol{b}$ и исследовании динамических систем с матрицей A. Если решения нет, то есть матрица A вырожденная, оказывается, что все равно можно пытаться сделать с этим уравнением что-то разумное. Для этого нужно заменить матрицу A на близкую к ней матрицу $A + \lambda\mathbb{I}$, как показано выше. При таком преобразовании вырожденная матрица обязательно снова станет невырожденной, «регулярной» — отсюда и «регуляризация». Кстати, с линейных уравнений на более общий случай операторов регуляризацию обобщил А. Н. Тихонов[1], и в его честь одну из форм регуляризации, к которой относится и гребневая регрессия, до сих пор называют «регуляризацией по Тихонову» (Tikhonov regularization).

В нашем примере при добавлении регуляризатора многочлены таковы:

$$\begin{aligned}
f_{\lambda=0}(x) = &-2{,}22 + 2{,}01x + 4{,}88x^2 + 31{,}13x^3 - 230{,}31x^4 + \\
&+ 103{,}72x^5 + 869{,}22x^6 - 966{,}67x^7 - 319{,}31x^8 + 505{,}64x^9, \\
f_{\lambda=0{,}01}(x) = &-2{,}32 + 3{,}40x - 2{,}33x^2 + 0{,}05x^3 - 0{,}51x^4 - \\
&- 0{,}29x^5 - 0{,}22x^6 - 0{,}06x^7 + 0{,}09x^8 + 0{,}24x^9, \\
f_{\lambda=1}(x) = &-2{,}46 + 1{,}45x - 0{,}19x^2 + 0{,}22x^3 - 0{,}13x^4 - \\
&- 0{,}05x^5 - 0{,}14x^6 - 0{,}13x^7 - 0{,}16x^8 - 0{,}16x^9.
\end{aligned}$$

Случай $\lambda = 0{,}0$ ничем не отличается от отсутствия регуляризации, при $\lambda = 1$ регуляризатор получается слишком сильным, а коэффициенты — слишком маленькими, а при $\lambda = 0{,}01$ получается «золотая середина». Вид кривых (рис. 2.4) подтверждает наши выводы; можно было бы выбрать новые точки в качестве валидационного множества и проверить полученные модели, но идея и так понятна.

---

[1] *Андрей Николаевич Тихонов* (1906–1993) — советский математик, академик АН СССР, основатель факультета вычислительной математики и кибернетики МГУ. Получил множество важных результатов в прикладной математике: разработал теорию однородных разностных схем для решения дифференциальных уравнений, а методы регуляризации получились из задач, связанных с поиском полезных ископаемых. А в 1948 году Тихонов организовал и возглавил вычислительную лабораторию, задачей которой был расчет процесса взрыва атомной бомбы.

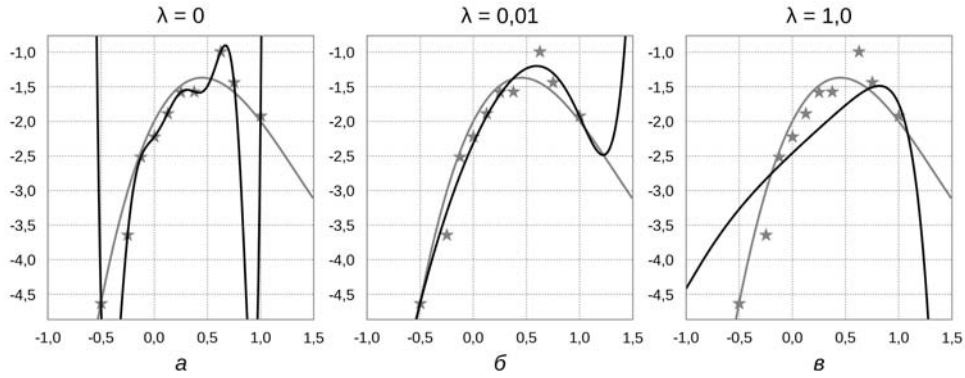


Рис. 2.4. Аппроксимация многочлена $f(x) = x^3 - 4x^2 + 3x - 2$ по десяти точкам с нормально распределенным шумом многочленами степени 9 с регуляризатором $\frac{\lambda}{2}\|\boldsymbol{w}\|^2$ для разных значений λ: *а* — $\lambda = 0$; *б* — $\lambda = 0{,}01$; *в* — $\lambda = 1$

Но пока регуляризация, хоть и работает, выглядит как в высшей степени *ad hoc* решение: мы взяли и по своей воле добавили лишний член в функцию ошибки. Как интерпретировать это новое слагаемое с вероятностных позиций?

Интуитивно смысл в следующем: мы решили, что в рассматриваемой модели «маловероятно», что у многочлена получатся большие коэффициенты. Иначе говоря, добавляя регуляризатор, мы пытались формализовать тот факт, что небольшие, короткие векторы коэффициентов более вероятны, чем длинные.

Чтобы объяснить эту формализацию, давайте посмотрим на регрессию с совсем байесовской стороны. До сих пор в нашем анализе линейной регрессии участвовало только правдоподобие. Теперь введем какое-нибудь априорное распределение, которое будет выражать наши априорные представления о том, какими должны быть веса регрессии $\boldsymbol{w}$. Начнем с того, что выберем этому распределению форму — пусть оно тоже будет нормальным: $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{\mu}_0, \Sigma_0)$. Рассмотрим снова набор данных $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ со значениями $\boldsymbol{t} = \{t_1, \ldots, t_N\}$. В этой модели мы по-прежнему предполагаем, что данные независимы и одинаково распределены:

$$p(\boldsymbol{t} \mid \mathcal{X}, \boldsymbol{w}, \sigma^2) = \prod_{n=1}^{N} \mathcal{N}\left(t_n \mid \boldsymbol{w}^\top \boldsymbol{x}_n, \sigma^2\right).$$

Теперь, чтобы найти апостериорное распределение, нам нужно подсчитать произведение плотностей нескольких нормальных распределений:

$$p(\boldsymbol{w} \mid \boldsymbol{t}) \propto p(\boldsymbol{t} \mid \mathcal{X}, \boldsymbol{w}, \sigma^2) p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{\mu}_0, \Sigma_0) \prod_{n=1}^{N} \mathcal{N}\left(t_n \mid \boldsymbol{w}^\top \boldsymbol{x}_n, \sigma^2\right).$$

Давайте это сделаем. Очевидно, что в показателе экспоненты (или в логарифме) снова получится квадратичная функция от $\boldsymbol{w}$, то есть произведение нормальных распределений снова будет нормальным:

$$p(\boldsymbol{w} \mid \boldsymbol{t}) = \mathcal{N}(\boldsymbol{w} \mid \boldsymbol{\mu}_N, \Sigma_N).$$

Чтобы подсчитать параметры $\boldsymbol{\mu}_N$ и Σ_N, придется немного попыхтеть, выделяя полный квадрат от $\boldsymbol{w}$. Оставим это упражнение читателю; результат же будет таким:

$$\boldsymbol{\mu}_N = \Sigma_N \left(\Sigma_0^{-1}\mu_0 + \frac{1}{\sigma^2} X^\top \boldsymbol{t} \right), \quad \Sigma_N = \left(\Sigma_0^{-1} + \frac{1}{\sigma^2} X^\top X \right)^{-1}.$$

А теперь давайте подсчитаем логарифм апостериорного распределения. Если мы возьмем априорное распределение с центром в нуле:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w} \mid \mathbb{O}, \frac{1}{\alpha}\mathbb{I}),$$

где $\mathbb{O}$ обозначает нулевую матрицу или вектор, а $\mathbb{I}$ — единичную матрицу, то от логарифма правдоподобия останется

$$\ln p(\boldsymbol{w} \mid \boldsymbol{t}) = -\frac{1}{2\sigma^2} \sum_{n=1}^{N} \left(t_n - \boldsymbol{w}^\top \boldsymbol{x}_n \right)^2 - \frac{\alpha}{2} \boldsymbol{w}^\top \boldsymbol{w} + \text{const}.$$

Получается, что мы пришли в точности к гребневой регрессии!

Обратите внимание: у нас получилось, что при нормальном априорном распределении $p(\boldsymbol{w})$ и правдоподобии нормально распределенного шума $p(\boldsymbol{t} \mid \mathcal{X}, \boldsymbol{w}, \sigma^2)$ с фиксированной дисперсией σ^2 логарифм апостериорного распределения $p(\boldsymbol{w} \mid \boldsymbol{t})$ тоже получился квадратичной функцией от $\boldsymbol{w}$, то есть $p(\boldsymbol{w} \mid \boldsymbol{t})$ — это тоже нормальное распределение! Это значит, что при фиксированной дисперсии и неизвестном среднем нормальное распределение является *самосопряженным*, то есть сопряженным самому себе.

На рис. 2.5 показан пример такого вычисления: умножая нормальное априорное распределение $p(\mu) = \mathcal{N}(\mu; 0, 1)$ на правдоподобие выпавшей точки 1 с нормально распределенным шумом с дисперсией $\frac{1}{2}$, мы получим апостериорное распределение

$$p(\mu \mid D) \propto \mathcal{N}(\mu; 0, 1)\mathcal{N}(1; \mu, \frac{1}{2}) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\mu^2} \times \frac{2}{\sqrt{2\pi}} e^{-\frac{4}{2}(1-\mu)^2} \propto$$
$$\propto e^{-\frac{1}{2}(\mu^2 + 4\mu^2 - 8\mu + 4)} = e^{-\frac{5}{2}\left(\mu^2 - \frac{8}{5}\mu + \frac{4}{5}\right)} \propto e^{-\frac{5}{2}\left(\mu - \frac{4}{5}\right)^2},$$

то есть апостериорное распределение тоже будет нормальным, и у него будет среднее $\frac{4}{5}$ и дисперсия $\frac{1}{\sqrt{5}}$.

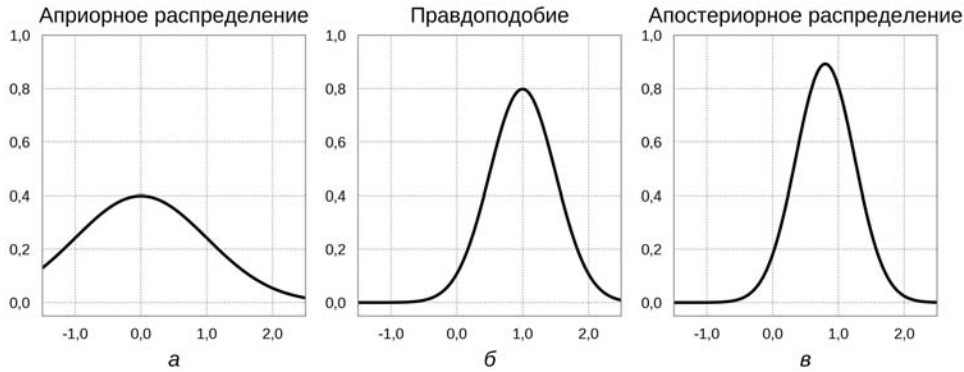


Рис. 2.5. Одна точка данных с нормально распределенным шумом: *а* — априорное распределение $p(\mu) = \mathcal{N}(\mu; 0, 1)$; *б* — правдоподобие $L(\mu) = \mathcal{N}\left(\mu; 1, \frac{1}{2}\right)$; *в* — апостериорное распределение $\mathcal{N}\left(\mu; \frac{4}{5}, \frac{1}{\sqrt{5}}\right)$

Можно теперь сделать и байесовские предсказания, но это уже, пожалуй, будет выходить за рамки книги. Нашей целью было познакомить читателя с основной идеей байесовского подхода к машинному обучению, а технические детали нам больше не понадобятся до самой главы 10, да и там они будут немного другими. Желающим все же освоить эту науку всерьез мы в качестве основных источников еще раз порекомендуем [44, 381], а сами будем двигаться дальше. В этом разделе мы говорили исключительно о регрессии, то есть о предсказании вещественного числа, функцией ошибки для которого часто вполне разумно взять просто сумму квадратов отклонений. Но не менее важны для нас будут и задачи классификации — а там с функцией ошибки дело обстоит немного хитрее.

2.3. Расстояние Кульбака — Лейблера и перекрестная энтропия

> Без идеального мужчины и идеальной женщины исследователь лишен твердого масштаба, который бы он мог применить к действительности, он ходит ощупью в темноте неизвестных, поверхностных суждений.
>
> *О. Вайнингер. Пол и характер*

Мы уже говорили о том, что большинство задач машинного обучения с учителем можно условно разделить на задачи регрессии, где целевая функция непрерывна, и задачи классификации, где целевая функция представляет собой выбор из

нескольких классов; их может быть много (как, например, в распознавании лиц, которое так хорошо делает Facebook), но они все-таки дискретны, и каждому из них должна соответствовать целая область в пространстве параметров.

Мы только что выяснили, что для задачи регрессии хорошей функцией ошибки является сумма квадратов отклонений предсказанных ответов от правильных. Эта функция соответствует нормально распределенному шуму, что для непрерывных величин более чем логично. Как выбрать функцию ошибки для задачи классификации? На первый взгляд кажется, что это совсем просто: если нам нужно отделить фотографии кошек от фотографий собак, давайте подсчитаем, сколько раз мы верно определили класс, а сколько раз неверно. То есть введем функцию ошибки, равную числу (или доле, что то же самое) верных ответов.

Такая метрика, которую называют *точностью* (accuracy) классификации, действительно часто представляет собой нашу конечную цель. Но вот беда: функция ошибки, которая просто подсчитывает число верных ответов, — это кусочно-постоянная функция. Она всегда локально постоянна, и маленькие изменения в классификаторе практически никогда не приведут к изменению ответа на каких-то тестовых примерах. Есть только некий конечный набор разделяющих поверхностей (множество меры нуль, как сказал бы математик), на которых значение функции внезапно и резко меняется.

И совершенно непонятно, как оптимизировать такую функцию. Градиентный спуск, который станет нашим основным инструментом для обучения нейронных сетей, здесь бесполезен, потому что производная функции равна нулю везде, кроме тех редких случаев, когда она и вовсе не существует.

Как же правильно выразить тот факт, что наши предсказания более или менее похожи на имеющиеся тренировочные данные, да еще и сделать это гладкой функцией, которую можно будет потом оптимизировать?

Из теории информации в информатику пришло понятие *относительной энтропии*, или *расстояния Кульбака — Лейблера* (Kullback — Leibler divergence, KL divergence, relative entropy), названного так в честь Соломона Кульбака[1] и Ричарда Лейблера. Расстояние Кульбака — Лейблера является по своей сути мерой разницы между двумя вероятностными распределениями P и Q. Как правило, считается, что распределение P — это «истинное» распределение, а Q — его приближение, и тогда расстояние Кульбака — Лейблера служит оценкой качества приближения.

---

[1] *Соломон Кульбак* (Solomon Kullback, 1907–1994) — американский математик и криптоаналитик. Первого апреля 1930 г. (характерная дата) Уильям Фридман, который в младенчестве эмигрировал из Кишинева в США из-за процветавшего в Российской империи антисемитизма, а в США стал знаменитым криптологом и провел всю жизнь на госслужбе, нанял «младшего криптоаналитика» Кульбака в числе трех первых сотрудников только что основанного Signals Intelligence Service (SIS). Во второй половине 1930-х годов Кульбак со своим другом Авраамом Синьковым (Abraham Sinkov) взломал коды тайной переписки японских дипломатов; эта работа стала еще более актуальной после Перл-Харбора. Знаменитая совместная работа с Ричардом Лейблером (Richard Leibler, 1914–2003) относится к 1951 году [287], а затем идея расстояния Кульбака — Лейблера была подробно изложена в книге Кульбака, вышедшей в 1959 году [286].

В теории информации оно является как раз количеством информации, которая теряется при приближении распределения P с помощью распределения Q.

Говоря формально, расстояние Кульбака — Лейблера от распределения P до распределения Q обозначается как $\mathrm{KL}\,(P\|Q)$ и определяется следующим образом:

$$\mathrm{KL}\,(P\|Q) = \int \log \frac{\mathrm{d}P}{\mathrm{d}Q}\mathrm{d}P,$$

где интеграл берется по всему пространству исходов, которое у P и Q должно быть общее. Нас, конечно, больше всего интересуют два частных случая:

- когда P и Q — дискретные случайные величины на дискретном множестве $X = \{x_1, \ldots, x_N\}$, расстояние Кульбака — Лейблера выглядит так:

$$\mathrm{KL}\,(P\|Q) = \sum_i p(x_i) \log \frac{p(x_i)}{q(x_i)},$$

  где $p(x_i)$ и $q(x_i)$ — собственно вероятности исхода x_i;
- когда P и Q — непрерывные случайные величины в пространстве $\mathbb{R}^d$, расстояние Кульбака — Лейблера можно записать как

$$\mathrm{KL}\,(P\|Q) = \int_{\mathbb{R}^d} p(\boldsymbol{x}) \log \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}\mathrm{d}\boldsymbol{x},$$

  где p и q — плотности распределений p и q.

Из формул сразу видно, что расстояние Кульбака — Лейблера на самом деле вовсе не является расстоянием![1] В частности, оно несимметрично: часто бывает, что $\mathrm{KL}\,(P\|Q) \neq \mathrm{KL}\,(Q\|P)$. Но для нас важнее другое свойство: расстояние Кульбака — Лейблера всегда неотрицательно, $\mathrm{KL}\,(P\|Q) \geq 0$, и оно равно нулю только тогда, когда распределения p и q совпадают почти всюду[2].

Как использовать расстояние Кульбака — Лейблера для задач классификации? Неформально говоря, мы будем пытаться подсчитать, насколько распределение на тестовых примерах, порожденное классификатором (назовем его q), похоже или непохоже на «истинное» распределение, задаваемое данными (назовем его p). А формально давайте начнем с бинарной классификации, где входные данные имеют вид $(\boldsymbol{x}, y)$ и y принимает только два значения; назовем их 0 и 1. Введем распределение данных достаточно тривиальным образом: $p(y = 1) = y$, а $p(y = 0) = 1 - y$; это значит, что в распределении данных все значения будут равны или 0, или 1. А распределение классификатора будет уж какое получится; классификатор пытается оценить вероятность положительного ответа $p(y \mid D, \boldsymbol{x})$, и именно ее мы и будем считать вероятностью $q(y)$.

[1] В английском языке здесь нет проблемы; по-русски тоже иногда говорят «дивергенция Кульбака — Лейблера» или «расхождение Кульбака — Лейблера», но «расстояние» используется чаще.

[2] Хорошее упражнение на понимание: попробуйте формально доказать это свойство.

Минимизировать будем не совсем расстояние Кульбака — Лейблера, а так называемую *перекрестную энтропию* (cross-entropy):

$$H(p, q) = \mathbb{E}_p [-\log q] = -\sum_y p(y) \log q(y).$$

Так будет удобнее для минимизации, а с расстоянием Кульбака — Лейблера перекрестная энтропия $H(p, q)$ связана самым прямым образом:

$$\mathrm{KL}\,(P\|Q) = \sum_y p(y) \log \frac{p(y)}{q(y)} = \\ = \sum_y p(y) \log p(y) - \sum_y p(y) \log q(y) = H(p) + H(p, q),$$

где $H(p)$ — энтропия распределения p. Получается, что для фиксированного p, задаваемого данными, нет разницы, что из этого минимизировать по q.

Для бинарной классификации целевая функция, которую мы будем минимизировать на наборе данных $D = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, обычно выглядит как средняя перекрестная энтропия по всем точкам в данных:

$$L(\boldsymbol{\theta}) = H(p_{\text{data}}, q(\boldsymbol{\theta})) = -\frac{1}{N} \sum_{i=1}^{N} \left(y_i \log \hat{y}_i(\boldsymbol{\theta}) + (1 - y_i) \log \left(1 - \hat{y}_i(\boldsymbol{\theta})\right)\right),$$

где $\hat{y}_i(\boldsymbol{\theta})$ — оценка вероятности ответа 1, полученная классификатором.

В результате мы получаем непрерывную функцию $L(\boldsymbol{\theta})$ от предсказанных классификатором вероятностей, которая оценивает, насколько хорошо он предсказывает метки в данных. $L(\boldsymbol{\theta})$ может служить функцией ошибки; ее можно дифференцировать, и вся оптимизация работает прекрасно.

Кстати, здесь получается, что классификатору нет смысла пытаться что-то «угадывать», если он не уверен. Например, если честно признаться, что мы ничего не поняли о вероятности ответа, $\hat{y}_i = \frac{1}{2}$, то ошибка на этом примере будет составлять $-\log \frac{1}{2} = \log 2$ независимо от правильного ответа. А если попытаться изобразить уверенность, подбросив монетку и указав полученному ею ответу вероятность q, то ожидаемая ошибка составит

$$\frac{1}{2} \log q + \frac{1}{2} \log(1 - q),$$

что всегда не меньше $\log 2$ и достигает этого минимума как раз в точке $q = \frac{1}{2}$. В частности, классификатору вообще никогда не стоит присваивать какому-то ответу абсолютную уверенность, $\hat{y} = 0$ или $\hat{y} = 1$, ведь в случае ошибки штраф будет буквально бесконечно большим!

С этой функцией ошибки тесно связана и классическая линейная модель классификации, которая называется *логистической регрессией*. Именно она обычно размещается на последнем уровне даже самых глубоких нейронных сетей: когда все признаки выделены, нужно в итоге на них все-таки сделать какой-то классификатор, и логистическая регрессия здесь подходит лучше всего. Поэтому для нас есть смысл изучить ее чуть подробнее.

Давайте рассматривать задачу классификации с вероятностной точки зрения: сопоставим каждому классу $\mathcal{C}_k$ плотность $p(\boldsymbol{x} \mid \mathcal{C}_k)$ (ее мы, конечно, заранее не знаем), определим некие априорные распределения $p(\mathcal{C}_k)$ (это по сути всего лишь размеры классов — насколько вероятно, что пример — из класса $\mathcal{C}_k$, если мы еще не знаем ничего о самом примере), а затем будем искать $p(\mathcal{C}_k \mid \boldsymbol{x})$ по теореме Байеса. Для двух классов получится:

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1) + p(\boldsymbol{x} \mid \mathcal{C}_2)p(\mathcal{C}_2)}.$$

Перепишем это равенство чуть по-другому:

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1) + p(\boldsymbol{x} \mid \mathcal{C}_2)p(\mathcal{C}_2)} = \frac{1}{1 + e^{-a}} = \sigma(a),$$

где

$$a = \ln \frac{p(\boldsymbol{x} \mid \mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x} \mid \mathcal{C}_2)p(\mathcal{C}_2)}, \qquad \sigma(a) = \frac{1}{1 + e^{-a}}.$$

Функция $\sigma(a)$ называется *логистическим сигмоидом*; это одна из классических функций активации для отдельных перцептронов, и мы о ней подробно поговорим в разделе 3.2.

Но сейчас нас больше интересует сама задача классификации. Логистическая регрессия — это модель, в которой мы напрямую делаем предположение о том, как выглядит аргумент сигмоида a. А именно, будем представлять a как линейную функцию от входных признаков: $a = \boldsymbol{w}^\top \boldsymbol{x}$. Дело в том, что сигмоид переводит любое вещественное число на отрезок $[0, 1]$; чем меньше аргумент, тем меньше результат (на минус бесконечности получается 0), и наоборот, на плюс бесконечности получается 1. Так что представлять сам ответ (номер класса, 0 или 1) в виде линейной функции было бы, понятное дело, довольно глупой идеей, но моделировать a линейной функцией уже вполне осмысленно, и даже, как видите, вполне логично интерпретировать результат как апостериорную вероятность того или иного класса.

В итоге получается, что

$$p(\mathcal{C}_1 \mid \boldsymbol{x}) = y(\boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}), \qquad p(\mathcal{C}_2 \mid \boldsymbol{x}) = 1 - p(\mathcal{C}_1 \mid \boldsymbol{x}),$$

и для обучения можно просто напрямую оптимизировать правдоподобие по $\boldsymbol{w}$.

Для входного набора данных $\{\boldsymbol{x}_n, t_n\}$, где $\boldsymbol{x}_n$ — входы, а t_n — соответствующие им правильные ответы, $t_n \in \{0,1\}$, мы получаем такое правдоподобие:

$$p(\boldsymbol{t} \mid \boldsymbol{w}) = \prod_{n=1}^{N} y_n^{t_n}(1-y_n)^{1-t_n}, \quad \text{где} \quad y_n = p(\mathcal{C}_1 \mid \boldsymbol{x}_n).$$

И теперь можно искать параметры максимального правдоподобия, максимизируя $\ln p(\boldsymbol{t} \mid \boldsymbol{w})$, то есть минимизируя следующую функцию:

$$E(\boldsymbol{w}) = -\ln p(\boldsymbol{t} \mid \boldsymbol{w}) = -\sum_{n=1}^{N} \left[t_n \ln y_n + (1-t_n)\ln(1-y_n)\right].$$

Обратите внимание: у нас опять получилась та же форма функции ошибки, что и выше! Это стандартная функция ошибки для задач классификации. Она также тривиальным образом обобщается на несколько классов: вместо логистического сигмоида будем теперь рассматривать так называемую *softmax*-функцию (сглаженный максимум, то есть на самом деле просто нормализованную экспоненту). Для K классов получается:

$$p(\mathcal{C}_k \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x} \mid \mathcal{C}_k)p(\mathcal{C}_k)}{\sum_{j=1}^{K} p(\boldsymbol{x} \mid \mathcal{C}_j)p(\mathcal{C}_j)} = \frac{e^{a_k}}{\sum_{j=1}^{K} e^{a_j}}.$$

Теперь у нас столько же аргументов, сколько классов: $a_k = \ln p(\boldsymbol{x} \mid \mathcal{C}_k)p(\mathcal{C}_k)$. И оптимизируем мы все ту же функцию правдоподобия, что и для двух классов. Давайте закодируем поступающие на вход правильные ответы в виде векторов длины K, в каждом из которых все компоненты равны нулю, кроме правильного класса, где стоит единица (это называется *one-hot* кодированием, и мы с ним не раз еще встретимся). Тогда для таких векторов $\boldsymbol{T} = \{\boldsymbol{t}_n\}$ правдоподобие выглядит так:

$$p(\boldsymbol{T} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) = \prod_{n=1}^{N}\prod_{k=1}^{K} p(\mathcal{C}_k \mid \boldsymbol{x}_n)^{t_{nk}} = \prod_{n=1}^{N}\prod_{k=1}^{K} y_{nk}^{t_{nk}},$$

где $y_{nk} = y_k(\boldsymbol{x}_n)$. И снова можно взять производную и прийти к тому же самому выражению, только в сумме станет больше слагаемых:

$$E(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) = -\ln p(\boldsymbol{T} \mid \boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}.$$

Понимать, как математически устроено представление задач классификации, крайне важно: большинство практических задач — это именно задачи классификации, и целевая функция в таком виде, как показано выше, будет постоянно встречаться как в этой книге, так и в вашей практике анализа данных. Однако здесь мы

не будем вдаваться в подробности обучения логистической регрессии. Все равно она для нас больше интересна как часть большой нейронной сети, верхний слой, так что и обучать ее мы будем вместе со всей остальной сетью с помощью одного из универсальных алгоритмов обучения. В наше время практически все эти алгоритмы представляют собой по сути модификации градиентного спуска, к которому мы незамедлительно и перейдем.

2.4. Градиентный спуск: основы

> Когда мы оторвались от верхнего пояса пены и очутились в бездне, нас сразу увлекло на очень большую глубину, но после этого мы спускались отнюдь не равномерно. Мы носились кругами, но не ровным, плавным движением, а стремительными рывками и толчками, которые то швыряли нас всего на какую-нибудь сотню футов, то заставляли лететь так, что мы сразу описывали чуть не полный круг. И с каждым оборотом мы опускались ниже, медленно, но очень заметно.
>
> *Э. А. По. Низвержение в Мальстрем*

Итак, мы выяснили, что основной нашей задачей и в машинном обучении вообще, и в обучении нейронных сетей в частности является уменьшение некоторой заданной функции ошибки. Теперь дело за малым — осталось понять, как же именно это сделать.

По сути это означает, что мы должны научиться решать *задачу оптимизации*: по заданной функции найти аргументы, в которых эта функция максимизируется или минимизируется (это, понятное дело, одно и то же, ведь перед функцией всегда можно просто поставить минус). Задачи и методы оптимизации — это, конечно, огромная тема, которой посвящены тысячи различных источников, и мы не надеемся рассмотреть ее подробно в этой книге. Дадим буквально пару ссылок на хорошие книги [68, 200, 435] и перейдем непосредственно к тому, что потребуется нам для оптимизации в нейронных сетях.

В случае последних дело существенно упрощается, потому что у нейронных сетей обычно нет никаких сложных ограничений на веса нейронов. Чаще всего сети построены так, что веса могут быть произвольными вещественными числами. Это значит, что многие сложности, связанные с формой допустимых множеств, краевыми эффектами и прочими лямбда-множителями Лагранжа, к нам не относятся. С другой стороны, дело немного усложняется тем, что сама функция, которую мы оптимизируем, нам обычно ни в каком разумном виде не задана. Легко было оптимизировать веса одного перцептрона — это была выпуклая задача, которую можно было решить практически в явном виде. Но как только мы переходим к оптимизации большой нейронной сети, сразу получается, что целевая функция — это очень

сложная композиция других функций, и кажется, что все, что мы можем сделать, — это вычислить функцию в той или иной точке, а с таким бедным «набором инструментов» решать оптимизационные задачи нелегко. Однако и здесь отчаиваться не стоит — по сравнению с самыми сложными задачами оптимизации у нас ситуация все-таки не настолько безнадежная. Хоть функция и действительно сложная, мы скоро увидим, что можем не только вычислить ее значение, но и *взять от нее производную* — а значит, для нас открыты все пути, связанные с градиентным спуском, о котором мы сейчас и поговорим.

Но сначала все же о неприятной новости. Хоть мы и научимся дифференцировать функцию ошибки в нейронных сетях, никакие силы уже не сделают ее *выпуклой*. Для выпуклых функций задача *локальной* оптимизации — найти локальный максимум, то есть «вершину», из которой все пути ведут вниз, — автоматически превращается в задачу *глобальной* оптимизации — найти точку, в которой достигается наибольшее значение функции, то есть самую высокую из таких «вершин».

Простейший пример выпуклой функции — это параболоид, функция второго порядка от своих аргументов. Пример такой функции мы видели, когда рассматривали линейную регрессию и функцию ошибки одного перцептрона:

$$E(w_1, \ldots, w_n) = -\frac{1}{2}\sum_{i=1}^{D}\left(y_i - \boldsymbol{w}^\top \boldsymbol{x}_i\right)^2.$$

Как найти максимум такого параболоида? Да просто взять производную, приравнять ее к нулю и решить полученное линейное уравнение. Заметьте, что уравнение линейное и решение у него будет одно-единственное. Экстремум тоже будет один, а потому автоматически глобальный. Иметь дело с выпуклыми функциями — вообще одно удовольствие, и нетривиальные задачи оптимизации для них возникают обычно либо при наличии хитрых ограничений, либо в постановке «как бы нам сойтись к экстремуму *поскорее*».

А у нейронных сетей, к сожалению (хотя почему «к сожалению» — так и должно быть, именно поэтому нейронные сети такие выразительные и могут решать так много разных задач), функция ошибки может задавать очень сложный ландшафт с огромным числом локальных максимумов и минимумов. Формально это всего лишь значит, что производная (градиент) обращается в ноль много раз в разных точках. А в реальности это значит, что даже если мы поднялись на вершину и удовлетворенно оглядели окрестности с высоты птичьего полета, мы никак не можем знать, действительно ли это самая высокая вершина или глобальный максимум находится в совершенно другом месте.

Заметим еще, что разных экстремумов может быть очень, *очень* много. Их легко может оказаться экспоненциально много от числа нейронов (то есть от числа аргументов функции, которую мы оптимизируем), и перечислить их все тоже нет совершенно никакой возможности. В нейронной сети этот эффект, кстати, очень легко продемонстрировать: представьте себе, что мы взяли нейроны одного

из внутренних слоев многослойной сети и поменяли их все местами, то есть заменили веса, ведущие в один нейрон и из него, на соответствующие веса другого нейрона. Совершенно очевидно, что в нейронной сети стандартной архитектуры от этого ровным счетом ничего не изменится: на следующий уровень придут ровно те же активации, что и раньше, ведь мы поменяли входные и выходные веса согласованным образом.

А это значит, что если у функции ошибки нейронной сети есть какой-то локальный максимум, то другой локальный максимум легко получить, просто переставив веса нейронов на внутреннем слое. Сколько таких перестановок? Правильно, $n!$, где n — число нейронов, которые мы переставляем; вот вам и экспоненциальное число локальных максимумов. Конечно, данный конкретный пример не очень содержателен: нам все равно, какой из эквивалентных максимумов выбрать; но существенно разных максимумов тоже может быть очень много.

Более того, из-за возможности оверфиттинга мы даже не можем быть уверены, что действительно хотим оказаться в этом глобальном экстремуме. Впрочем, этот вопрос решается скорее через регуляризацию, о которой мы еще не раз вспомним в книге. Но, хотя гарантировать точное решение задачи оптимизации оказывается невозможно, мы все-таки можем попытаться найти наилучший вариант из тех, что нам доступны, и постараться выбрать все-таки не первый попавшийся, а какой-нибудь «неплохой» максимум функции. Для этого и предназначены так называемые *эвристические* методы оптимизации, которые мы будем рассматривать дальше в этой главе. Главный из них — *градиентный спуск*; собственно, почти все современные методы оптимизации невыпуклых функций представляют собой те или иные его варианты.

Сделаем в этом месте небольшое лирическое отступление. Понятно, что, формально говоря, применение градиентного спуска возможно только на дифференцируемых функциях, где производная будет известна в любой точке. Но в жизни, конечно, иногда попадаются и недифференцируемые функции, а иногда — функции с тривиальными производными. Например, в обучении с подкреплением известно только текущее значение функции ценности. А в *обучении ранжированию*, когда мы хотим получить на выходе некое упорядочивание объектов по какому-то критерию, например в выдаче поисковика или рекомендательной системы, оказывается, что градиент функции ошибки почти всюду равен нулю!

Действительно, представьте себе поисковую выдачу, упорядоченную по релевантности документа запросу. Релевантность — это и есть функция, которую мы приближаем моделью (например, нейронной сетью). Но в случае обучения ранжированию небольшие изменения в весах приводят к небольшим изменениям релевантности каждого документа, и в результате сравнительный порядок почти никогда не меняется при малых изменениях весов. Но целевая функция зависит только от порядка! Получается, что функция ошибки кусочно-постоянна, состоит из большого числа маленьких горизонтальных «островов», и градиент от нее абсолютно бесполезен: он либо равен нулю, либо не существует.

Что же делать в таких случаях? Оказывается, что градиентный спуск — это настолько всеобъемлющий метод, что альтернатив ему совсем мало. Методы оптимизации в основном занимаются тем, что ускоряют градиентный спуск, разрабатывают разные его варианты, добавляют ограничения, но принципиально других методов почти нет. Поэтому, когда градиентный спуск неприменим, решение состоит в том, чтобы все-таки именно его и применить. В зависимости от того, в чем состоит сложность, можно воспользоваться такими приемами:

1) если производная есть, но вычислить ее не получается, а можно считать только значения функций в разных точках, то производную можно подсчитать приближенно; например, прямо по определению:

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon},$$

то есть, если взять достаточно маленький ϵ, производную можно приближенно подсчитать по этой самой формуле; а если вы когда-нибудь слушали курс под названием «Методы вычислений» или подобный, вам могут быть знакомы и чуть более сложные, но и более точные формулы конечных разностей, например:

$$f'(x) \approx \frac{f(x-2\epsilon) - 8f(x-\epsilon) + 8f(x+\epsilon) - f(x+2\epsilon)}{12\epsilon};$$

все это большая наука, и переход от производной функции одного аргумента к градиенту тоже не вполне тривиален, но основная идея здесь именно в том, что даже если мы умеем только вычислять функцию, все равно производную, а значит, и градиент, можно вычислить приближенно;

2) если производной совсем нет и не предвидится, то приближать, скорее всего, придется саму функцию, которую мы пытаемся оптимизировать; можно попробовать выбрать достаточно «хорошо себя ведущую» функцию, которая, тем не менее, похожа на ту, что надо оптимизировать;

3) если производная тривиальна, как в случае обучения ранжированию, то подлежащую оптимизации функцию тоже приходится модифицировать; при обучении ранжированию, например, мы будем оптимизировать не саму функцию качества, зависящую только от расположения элементов в списке, а некую специальную гладкую функцию ошибки, которая будет тем больше, чем «более ошибочно» окажется вычисление релевантности; нечто подобное мы уже видели, когда обучали один перцептрон, где вместо кусочно-постоянной функции ошибки классификации пришлось взять функцию, зависящую от того, насколько сильно ошибся классификатор.

Но мы отвлеклись. Итак, получается, что при градиентном спуске больше всего изменяется тот параметр, который имеет самую большую по модулю производную.

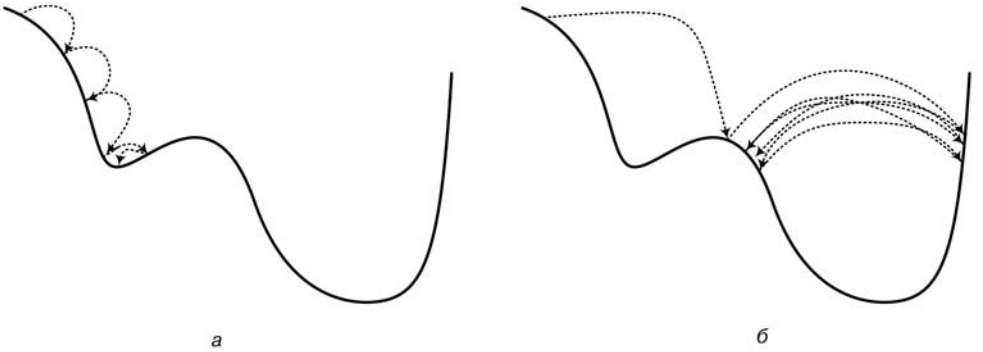


Рис. 2.6. Проблемы со скоростью градиентного спуска:
а — слишком маленькие шаги;
б — слишком большие шаги

Это значит, что поверхность наклонена в этом направлении больше, чем во всех остальных. Вообще, градиентный спуск — это в точности движение в направлении по той касательной к поверхности функции, которая наклонена вниз сильнее всего. Аналогия со скатывающимся вниз шариком здесь абсолютно уместна, шарик будет катиться именно в эту сторону (а вот скорость у него будет меняться несколько иначе, об этом мы еще поговорим).

Цель такого движения состоит в том, чтобы скатиться по поверхности функции к некоторому *локальному минимуму*, точке экстремума функции, которую мы оптимизируем. Поскольку в момент, когда мы окажемся в локальном минимуме, градиент будет нулевым, шаги градиентного спуска, которые связаны с абсолютным значением градиента, тоже будут постепенно уменьшаться и в самой точке экстремума спуск остановится вовсе. Конечно, от дискретной алгоритмической процедуры градиентного спуска точного попадания в ноль мы не дождемся, но в тот момент, когда изменения станут совсем маленькими, мы можем смело останавливать алгоритм градиентного спуска.

Единственный (пока что!) параметр градиентного спуска — это его *скорость обучения* η; она регулирует размер шага, который мы делаем в направлении склона-градиента. В чистом градиентном спуске скорость обучения задается вручную, и она может достаточно сильно повлиять на результат. В частности, могут возникнуть две противоположные друг другу проблемы (рис. 2.6):

- если шаги будут слишком маленькими, то обучение будет слишком долгим, и повышается вероятность застрять в небольшом неудачном локальном минимуме по дороге (см. рис. 2.6, *а*);
- а если слишком большими, можно бесконечно прыгать через искомый минимум взад-вперед, но так и не прийти в самую нижнюю точку (см. рис. 2.6, *б*).

Оставшаяся часть главы будет посвящена тому, как бороться с этими двумя эффектами. Но сначала — последнее, но тем не менее крайне важное замечание[1].

Давайте посмотрим, как выглядит градиент функции ошибки в случае какой-нибудь реалистичной задачи машинного обучения. Предположим, что у нас есть набор данных D, состоящий из пар $(\boldsymbol{x}, y)$, где $\boldsymbol{x}$ — признаки, а y — правильный ответ. Модель с весами $\boldsymbol{\theta}$ на этих данных делает некоторые предсказания $f(\boldsymbol{x}, \boldsymbol{\theta})$, $\boldsymbol{x} \in D$, и задана функция ошибки E, которую можно подсчитать на каждом примере, $E(f(\boldsymbol{x}, \boldsymbol{\theta}), y)$. Например, это может быть квадрат или модуль отклонения $f(\boldsymbol{x}, \boldsymbol{\theta})$ от y в случае регрессии или перекрестная энтропия в случае классификации. В любом случае общая функция ошибки будет суммой ошибок на каждом тренировочном примере:

$$E(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},y)\in D} E(f(\boldsymbol{x}, \boldsymbol{\theta}), y).$$

А градиентный спуск будет выглядеть так:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla E(\boldsymbol{\theta}_{t-1}) = \boldsymbol{\theta}_{t-1} - \eta \sum_{(\boldsymbol{x},y)\in D} \nabla E(f(\boldsymbol{x}, \boldsymbol{\theta}_{t-1}), y).$$

Для того чтобы сделать один-единственный шаг градиентного спуска, нужно, получается, пробежаться по всему тренировочному множеству! А оно может быть, и так часто бывает в реальных задачах, гигантским. Неужели получается, что градиентный спуск на практике не работает?

На самом деле, увы, действительно не работает. Работает не простой, а *стохастический* градиентный спуск, в котором ошибка подсчитывается и веса подправляются не после прохода по всему тренировочному множеству, а после каждого примера:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla E(f(\boldsymbol{x}_t, \boldsymbol{\theta}_{t-1}), y_t).$$

Основное преимущество стохастического градиентного спуска состоит в том, что ошибка на каждом шаге считается быстро, веса меняются сразу же, что очень сильно ускоряет обучение. На практике нередки ситуации, когда обучение фактически уже сошлось еще до того, как мы даже один раз пробежались по всем тренировочным примерам! Но кроме вычислительных, есть и содержательные преимущества: стохастический градиентный спуск работает «более случайно», чем обычный, и поэтому можно надеяться, что он не остановится в маленьких локальных минимумах, а найдет впадину посерьезнее. Вообще, в локальной оптимизации часто помогает пытаться сделать обновления «как можно более случайными», стараться как можно больше исследовать в пространстве поиска.

Однако стохастический градиентный спуск — тоже не предел мечтаний: обновлять веса модели после каждого тренировочного примера зачастую выходит

[1] Жаль, нет в русском языке столь же употребительного аналога last but not least.

чересчур накладно. Поэтому на практике обычно используется нечто среднее: стохастический градиентный спуск по *мини-батчам* (mini-batch), небольшим подмножествам тренировочного набора. Это позволяет сохранить все плюсы стохастического градиента (несколько десятков или сотен примеров — это обычно очень малая часть тренировочного множества, так что все, что мы говорили о маленьких локальных минимумах и скорости обучения, сохраняет силу), но при этом пользоваться процедурами матричной арифметики, которые очень быстро вычисляются на видеокарте. Во многих практических примерах в этой книге мы будем пользоваться мини-батчами.

2.5. Граф вычислений и дифференцирование на нем

> В каком-то уголке Швейцарии жил старый граф, у которого был только один сын, да и тот такой несмышленый, что, как ни учили его, никак и ничему не могли его научить.
>
> *Братья Гримм. Три языка*

> Таково определение графа. Но, как и столь презираемое им определение чистых эстетиков, оно с одной стороны — беспредельно, с другой — односторонне. В свое определение граф впихнул все — до анекдота включительно, и не дал самого существенного признака искусства, который заключается в *творческом* начале. Без *творчества* нет искусства.
>
> *А. И. Богданович. Граф Толстой об искусстве и науке*

В этом разделе мы введем основополагающее понятие для реализации алгоритмов обучения нейронных сетей. Оказывается, что если у нас получится представить сложную функцию как композицию более простых, то мы сможем и эффективно вычислить ее производную по любой переменной, что и требуется для градиентного спуска. Самое удобное представление в виде композиции — это представление в виде *графа вычислений*. Граф вычислений — это граф[1], узлами которого являются функции (обычно достаточно простые, взятые из заранее фиксированного набора), а ребра связывают функции со своими аргументами.

Это проще увидеть своими глазами, чем формально определять; посмотрите на рис. 2.7, где показаны три графа вычислений для одной и той же функции:

$$f(x, y) = x^2 + xy + (x + y)^2.$$

На рис. 2.7, *а* граф получился совсем прямолинейный, потому что мы разрешили использовать в качестве вершин унарную функцию «возведение в квадрат».

[1] Да, это нетривиальное замечание; в математике иногда бывает так, что белая лошадь оказывается вовсе не лошадью...

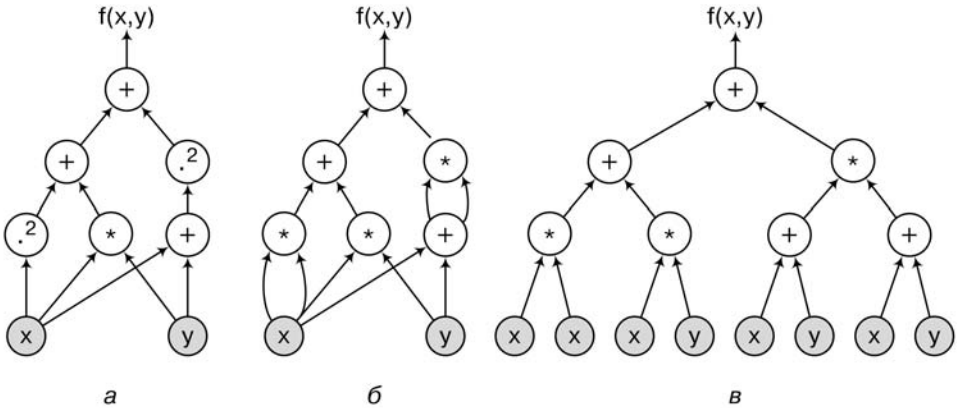


Рис. 2.7. Графы вычислений для функции $f(x, y) = x^2 + xy + (x + y)^2$:
а — с использованием функций $+$, $\times$ и ^{2}; *б* — с использованием функций $+$ и $\times$;
в — в виде дерева с использованием функций $+$ и $\times$

А на рис. 2.7, *б* граф чуть более хитрый: явное возведение в квадрат мы заменили обычным умножением. Впрочем, он от этого не сильно увеличился в размерах, потому что в графе вычислений на рис. 2.7, *б* разрешается по несколько раз переиспользовать результаты предыдущих вычислений и достаточно просто подать один и тот же x два раза на вход функции умножения, чтобы вычислить его квадрат. Для сравнения мы нарисовали на рис. 2.7, *в* граф той же функции, но без переиспользования; теперь он становится деревом, но в нем приходится повторять целые большие поддеревья, из которых раньше могли выходить сразу несколько путей к корню дерева[1].

В нейронных сетях в качестве базисных, элементарных функций графа вычислений обычно используют функции, из которых получаются нейроны. Например, скалярного произведения векторов $\boldsymbol{x}^\top \boldsymbol{y}$ и логистического сигмоида σ достаточно для того, чтобы построить любую, даже самую сложную нейронную сеть, составленную из нейронов с функцией активации σ. Можно было бы, кстати, выразить скалярное произведение через сложение и умножение, но слишком «мельчить» тут тоже не обязательно: элементарной может служить любая функция, для которой мы сможем легко вычислить ее саму и ее производную по любому аргументу. В частности, прекрасно подойдут любые функции активации нейронов, о которых мы будем подробно говорить в разделе 3.3.

[1] В теории сложности на самом деле рассматривают обе модели. Если переиспользовать результаты можно и в итоге может получиться любой направленный ациклический граф, это схемная сложность, в которой вычисление функции представляется булевой схемой. А если нельзя и граф должен быть деревом, то получается формульная сложность. Она соответствует линейному размеру формулы, которой можно записать функцию — внутри формулы ведь не получится переобозначить большой кусок новой переменной.

Итак, мы поняли, что многие математические функции, даже с очень сложным поведением, можно представить в виде графа вычислений, где в узлах стоят элементарные функции, из которых, как из кирпичиков, получается сложная композиция, которую мы и хотим подсчитать. Собственно, с помощью такого графа даже с не слишком богатым набором элементарных функций можно приблизить любую функцию сколь угодно точно; об этом мы еще поговорим чуть позже.

А сейчас вернемся к нашему основному предмету — машинному обучению. Как мы уже знаем, цель машинного обучения — подобрать модель (чаще всего здесь имеются в виду веса модели, заданной в параметрическом виде) таким образом, чтобы она лучше всего описывала данные. Под «лучше всего» здесь, как правило, имеется в виду оптимизация некоторой функции ошибки. Обычно она состоит из собственно ошибки на обучающей выборке (функции правдоподобия) и регуляризаторов (априорного распределения), но сейчас нам достаточно просто считать, что есть некая довольно сложная функция, которая дана нам свыше, и мы хотим ее минимизировать. Как мы уже знаем, один из самых простых и универсальных методов оптимизации сложных функций — это градиентный спуск. Мы также недавно выяснили, что один из самых простых и универсальных методов задать сложную функцию — это граф вычислений.

Оказывается, градиентный спуск и граф вычислений буквально созданы друг для друга! Как вас наверняка учили еще в школе (школьная программа вообще содержит много неожиданно глубоких и интересных вещей), чтобы вычислить производную композиции функций (в школе это, вероятно, называлось «производная сложной функции», как будто у слова «сложный» без этого недостаточно много значений), достаточно уметь вычислять производные ее составляющих:

$$(f \circ g)'(x) = (f(g(x)))' = f'(g(x))g'(x).$$

Интуиция здесь простая: если f зависит от g, а g зависит от x, то маленькое изменение параметра x на δx приведет к маленькому изменению значения $g(x)$ примерно на $\delta g = g'(x)\delta x$. А оно, в свою очередь, приведет к маленькому изменению значения $f(g(x))$ примерно на $\delta f = f'(g(x))\delta g = f'(g(x))g'(x)\delta x$.

По сравнению со школьной программой нам потребуется только один вполне естественный дополнительный шаг, который делается обычно уже в программе университетских курсов: нам нужно будет понять, что происходит, если f, g и x — это не скалярные, а векторные величины. С f все понятно, можно просто брать производные отдельно по каждой компоненте вектора; так что дальше будем считать, что f — это скалярная функция. Если x — это вектор $\boldsymbol{x} = (x_1, \ldots, x_n)$, то вместо частной производной мы говорим о *градиенте*, векторе частных производных:

$$\nabla_{\boldsymbol{x}} f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

Правило применяется для каждой компоненты отдельно, и результат выходит опять вполне ожидаемый:

$$\nabla_{\boldsymbol{x}}(f\circ g)=\begin{pmatrix}\frac{\partial f\circ g}{\partial x_1}\\ \vdots \\ \frac{\partial f\circ g}{\partial x_n}\end{pmatrix}=\begin{pmatrix}\frac{\partial f}{\partial g}\frac{\partial g}{\partial x_1}\\ \vdots \\ \frac{\partial f}{\partial g}\frac{\partial g}{\partial x_n}\end{pmatrix}=\frac{\partial f}{\partial g}\nabla_{\boldsymbol{x}} g.$$

А вот если g — это вектор, то получается, что f зависит от x не в одном месте, а сразу в нескольких, $f=f(g_1(x),g_2(x),\ldots,g_k(x))$. Говоря нестрого, это значит, что малое приращение δx превратится в несколько разных приращений $\delta g_1,\ldots,\delta g_k$, и каждое из них внесет свой собственный вклад в приращение функции f, а именно, $\delta f=\frac{\partial f}{\partial g_1}\delta g_1+\ldots+\frac{\partial f}{\partial g_k}\delta g_k$. Говоря формально:

$$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial g_1}\frac{\partial g_1}{\partial x}+\ldots+\frac{\partial f}{\partial g_k}\frac{\partial g_k}{\partial x}=\sum_{i=1}^{k}\frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}.$$

С градиентами все точно так же:

$$\nabla_{\boldsymbol{x}} f=\frac{\partial f}{\partial g_1}\nabla_{\boldsymbol{x}} g_1+\ldots+\frac{\partial f}{\partial g_k}\nabla_{\boldsymbol{x}} g_k=\sum_{i=1}^{k}\frac{\partial f}{\partial g_i}\nabla_{\boldsymbol{x}} g_i.$$

Обратите внимание, что у нас получилась формула матричного умножения:

$$\nabla_{\boldsymbol{x}} f=\nabla_{\boldsymbol{x}\boldsymbol{g}}\nabla_{\boldsymbol{g}} f,\quad\text{где}\quad\nabla_{\boldsymbol{x}\boldsymbol{g}}=\begin{pmatrix}\frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_k}{\partial x_1}\\ \vdots & & \vdots \\ \frac{\partial g_1}{\partial x_n} & \cdots & \frac{\partial g_k}{\partial x_n}\end{pmatrix}.$$

Такая матрица частных производных называется *матрицей Якоби*[1], а ее определитель — *якобианом*; они еще не раз нам встретятся. Теперь мы можем подсчитать производные и градиенты любой композиции функций, в том числе векторных, и для этого нужно только уметь вычислять производные каждой компоненты. Для графа все это де факто сводится к простой, но очень мощной идее: если мы знаем граф вычислений и знаем, как брать производную в каждом узле, этого достаточно, чтобы взять производную от всей сложной функции, которую задает граф!

---

[1] *Карл Густав Якоб Якоби* (Carl Gustav Jacob Jacobi, 1804–1851) — немецкий математик и механик. Основной темой его математических трудов была теория эллиптических функций: начинал он с того, что разработал теорию тета-функций Якоби, матрица вторых частных производных не раз встречается в его работах по вариационному исчислению, а применения эллиптических функций к алгебре и теории чисел привели к тому, что Якоби стал одним из основателей теории определителей. Его родным братом был Мориц Герман Якоби, изобретатель электродвигателя и гальванопластики, который у нас более известен как Борис Семенович, потому что большую часть научной карьеры провел в России.

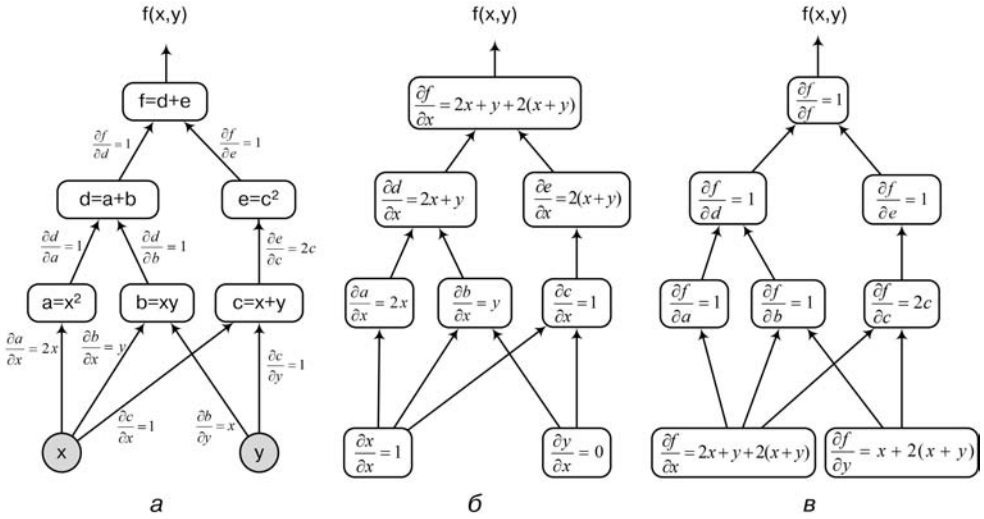


Рис. 2.8. Как брать производную на графе вычислений для функции $f(x, y) = x^2 + xy + (x + y)^2$: *а* — граф вычислений и частные производные; *б* — берем производную $\frac{\partial f}{\partial x}$ методом прямого распространения; *в* — берем частные производные f по всем переменным методом обратного распространения

Давайте сначала разберем это на примере: рассмотрим все тот же граф вычислений, который был показан на рис. 2.7. На рис. 2.8, *а* показаны составляющие граф элементарные функции; мы обозначили каждый узел графа новой буквой, от a до f, и выписали частные производные каждого узла по каждому его входу.

Теперь можно подсчитать частную производную $\frac{\partial f}{\partial x}$ так, как показано на рис. 2.8, *б*: начинаем считать производные с истоков графа и пользуемся формулой дифференцирования композиции, чтобы подсчитать очередную производную. Например:

$$\frac{\partial d}{\partial x} = \frac{\partial d}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial d}{\partial b}\frac{\partial b}{\partial x} = 1 \cdot 2x + 1 \cdot y = 2x + y.$$

Но можно пойти и в обратном направлении, как показано на рис. 2.8, *в*. В этом случае мы начинаем с истока, где всегда стоит частная производная $\frac{\partial f}{\partial f} = 1$, а затем разворачиваем узлы в обратном порядке по формуле дифференцирования сложной функции.

Формула окажется здесь применима точно так же; например, в самом нижнем узле мы получим:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial x} = 2x + y + 2(x + y).$$

Таким образом, можно применять формулу дифференцирования композиции на графе либо от истоков к стокам, получая частные производные каждого узла по одной и той же переменной $\frac{\partial x}{\partial x}, \frac{\partial a}{\partial x}, \ldots, \frac{\partial f}{\partial x}$, либо от стоков к истокам, получая частные производные стоков по всем промежуточным узлам $\frac{\partial f}{\partial f}, \frac{\partial f}{\partial e}, \ldots, \frac{\partial f}{\partial x}$. Конечно, на практике для машинного обучения нам нужен скорее второй вариант, чем первый: функция ошибки обычно одна, и нам требуются ее частные производные сразу по многим переменным, в особенности по всем весам, по которым мы хотим вести градиентный спуск.

В общем виде алгоритм такой: предположим, что нам задан некоторый направленный ациклический граф вычислений $G = (V, E)$, вершинами которого являются функции $g \in V$, причем часть вершин соответствует входным переменным $x_1, \ldots, x_n$ и не имеет входящих ребер, одна вершина не имеет исходящих ребер и соответствует функции f (весь граф вычисляет эту функцию), а ребра показывают зависимости между функциями, стоящими в узлах. Тогда мы уже знаем, как получить функцию f, стоящую в «последней» вершине графа: для этого достаточно двигаться по ребрам и вычислять каждую функцию в топологическом порядке.

А чтобы узнать частные производные этой функции, достаточно двигаться *в обратном направлении*. Если нас интересуют частные производные функции $f \in V$, то в полном соответствии с формулами выше мы можем подсчитать $\frac{\partial f}{\partial g}$ для каждого узла $g \in V$ таким образом:

- сначала инициализируем $\frac{\partial f}{\partial f} = 1$;
- затем для каждой вершины $g \in V$, у которой все *дети*, то есть вершины, в которые идут из нее ребра, уже обработаны алгоритмом, вычислим

$$\frac{\partial f}{\partial g} = \sum_{g' \in \text{Children}(g)} \frac{\partial f}{\partial g'} \frac{\partial g'}{\partial g}.$$

Вот и все! Когда мы дойдем до истоков графа, до вершин $x_1, \ldots, x_n$, мы получим частные производные $\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}$, то есть как раз вычислим градиент $\nabla_{\boldsymbol{x}} f$.

Такой подход называют алгоритмом *обратного распространения* (backpropagation, backprop, bprop), потому что частные производные считаются в направлении, обратном ребрам графа вычислений. А алгоритм вычисления самой функции или производной по одной переменной, как на рис. 2.8, *б*, называют алгоритмом *прямого распространения* (forward propagation, fprop).

И последнее важное замечание: обратите внимание, что за все то время, пока мы обсуждали графы вычислений, дифференциалы, градиенты и тому подобное, мы, собственно, ни разу всерьез не упомянули нейронные сети! И действительно, метод вычисления производных/градиентов по графу вычислений сам по себе совершенно никак не связан с нейронными сетями. Это полезно иметь в виду, особенно в делах практических, к которым мы перейдем уже в следующем разделе. Дело в том, что библиотеки Theano и TensorFlow, которые мы будем обсуждать ниже

и на которых делается большая часть глубокого обучения, — это, вообще говоря, библиотеки для *автоматического дифференцирования*, а не для обучения нейронных сетей. Все, что они делают, — позволяют вам задать граф вычислений и чертовски эффективно, с распараллеливанием и переносом на видеокарты, вычисляют градиент по этому графу.

Конечно, «поверх» этих библиотек можно реализовать и собственно библиотеки со стандартными конструкциями нейронных сетей, и это люди тоже постоянно делают (мы ниже будем рассматривать Keras), но важно не забывать базовую идею автоматического дифференцирования. Она может оказаться гораздо более гибкой и богатой, чем просто набор стандартных нейронных конструкций, и может случиться так, что вы будете крайне успешно использовать TensorFlow вовсе не для обучения нейронных сетей.

2.6. И о практике: введение в TensorFlow и Keras

> Она плохо сознавала, что делает и что должна сделать, но вместе с тем отлично знала, что должна все устроить, и устроить сейчас же. В ней билась практическая бабья сметка.
>
> *Д. Н. Мамин-Сибиряк. Три конца*

В последние годы обучение нейронных сетей превратилось в высшей степени инженерную дисциплину. Появилось несколько очень удобных библиотек, которые позволяют буквально в пару строк кода построить модель нейронной сети (сколь угодно глубокой), сформировать для нее граф вычислений, автоматически подсчитать градиенты и выполнить процесс обучения, причем сделать это можно как на процессоре, так и — совершенно прозрачным образом, изменив пару строк или пару ключей при запуске — на видеокарте, что обычно ускоряет обучение в десятки раз. С каждым годом эти библиотеки становятся все удобнее, выходят новые версии существующих, а также абсолютно новые программные продукты. Есть и библиотеки общего назначения, которые могут создать любой граф вычислений, и специализированные надстройки, которые реализуют разные компоненты нейронных сетей: обычные слои, сверточные, рекуррентные, рекуррентные слои из LSTM или GRU, современные алгоритмы оптимизации... Все это обычно можно «пощупать руками», реализовать и обучить в домашних условиях, без долгих и мучительных процессов разработки.

Поэтому современная книга о глубоком обучении не может обойтись без примеров конкретных программных реализаций и кода: в конце концов, мы пишем эту книгу для того, чтобы вы смогли не только разобраться в паре новых для себя абстракций, но и начать реально использовать нейронные сети в работе. Мы отдаем себе отчет в том, что эти разделы книги наиболее преходящи и быстрее всего

устареют, и, конечно же, не советуем нашим читателям из 2020-х годов и позже[1] пользоваться книгой как практическим руководством по библиотекам TensorFlow или Keras, хотя общие идеи приведенных здесь реализаций вряд ли изменятся.

И еще одно замечание. Все библиотеки, которыми мы пользуемся для реализации нейронных сетей в этой книге, имеют основной интерфейс к языку программирования Python, и все примеры кода тоже будут на Python. Этот язык стал де-факто стандартом в современном машинном обучении и обработке данных[2], хотя в нейронных сетях есть и другие примеры — например, библиотека Torch предлагает описывать структуру сетей в виде скриптов на языке Lua[3]. Заметим, что это не означает, что сами вычисления в библиотеках целиком написаны на Python — конечно, для оптимизации низкоуровневых вычислений пишут на C и обращаются к еще более низкоуровневым библиотекам — драйверам видеокарты и особенно библиотеке cuDNN [102]. Мы не можем включать в книгу еще и руководство по языку Python и в дальнейшем будем просто предполагать, что сам язык и основная его вычислительная библиотека NumPy вам знакомы. Если же это не так, то очень рекомендуем познакомиться с Python; в этом вам могут помочь, например, пособия [126, 423, 491] и многочисленные онлайн-ресурсы для обучения Python.

Среди библиотек общего назначения, которые способны строить граф вычислений и проводить автоматическое дифференцирование, долгое время бесспорным лидером была Theano, разработанная в университете Монреаля, в группе классика глубокого обучения Йошуа Бенджи (Yoshua Bengio) [528, 529]. Однако в ноябре 2015 года Google выпустила (с открытым исходным кодом) библиотеку TensorFlow [523], предназначенную для того же самого. TensorFlow стала вторым поколением библиотек глубокого обучения в Google; она была призвана заменить библиотеку DistBelief [295], которая послужила многим выдающимся результатам в истории глубокого обучения, но так и осталась проприетарной. Библиотеки Theano и TensorFlow на данный момент остаются двумя бесспорными лидерами в этой области, и сложно уверенно рекомендовать одну из них. Для книги мы выбрали именно TensorFlow, так как можно ожидать, что мощь разработчиков Google в итоге приведет к тому, что новые интересные возможности будут появляться в TensorFlow быстрее, чем в Theano. Время покажет, оправдается ли наше предположение.

Основная абстракция, которая потребуется нам для того, чтобы понять все, что происходит в коде таких библиотек, как TensorFlow или Theano, — это все тот же граф вычислений, который мы рассматривали в предыдущем разделе. Программа, использующая TensorFlow, обычно просто задает граф вычислений,

---

[1] Привет вам, коллеги, от пионеров, то есть, простите, информатиков прошлого! Очень надеемся, что вы существуете и книга эта не была напрочь забыта сразу после выхода.

[2] Опять же, в нашей области все меняется очень быстро, так что вам, коллеги из будущего, может быть виднее.

[3] Недавно, впрочем, появилась библиотека PyTorch, которая позволяет писать на Python, используя Torch в качестве бэкенда.

а потом запускает процедуру вроде `session.run`, которая выполняет вышеописанные вычисления и получает собственно результаты.

TensorFlow при этом обратится к тому или иному бэкенду (backend), низкоуровневой библиотеке, которая собственно и будет запускать код вычислений. Как и любая современная библиотека для символьного дифференцирования и/или обучения нейронных сетей, TensorFlow может работать как на процессоре, так и на видеокарте. Однако, в отличие от многих других библиотек, TensorFlow, будучи детищем Google, умеет еще и «из коробки» распараллеливать обучение на распределенные кластеры компьютеров; в книге, правда, таких примеров не будет.

Пожалуй, основная критика, с которой сталкивается библиотека TensorFlow, — это то, что она до сих пор не самая быстрая среди всех аналогичных; например, библиотека Torch [89, 90] на данный момент (конец 2016 года) считается более эффективной. Но, во-первых, мы не зря написали «на данный момент» и указали время: TensorFlow очень быстро развивается, и ускорение бэкенда, конечно, является одной из основных задач. А во-вторых, даже если вдруг появится еще более быстрая библиотека для вычислений на графах, ваши труды по изучению TensorFlow все равно не пропадут даром, ведь «фронтенд», в котором вы определяете граф вычислений, и бэкенд, который их собственно производит, вполне можно разделить. Например, TinyFlow [533] — это библиотека, позволяющая конструировать граф вычислений на TensorFlow, а обучать его на том же самом Torch. Получается нечто похожее на то, что в теории компиляторов называют LLVM (Low Level Virtual Machine, низкоуровневая виртуальная машина): библиотека получает описание графа вычислений на одном «языке», конвертирует его в некое внутреннее представление, а потом переводит в нужный формат и отправляет на вход бэкенду, который может быть написан на совершенно другом «языке».

Итак, давайте знакомиться с TensorFlow. Основной объект, которым оперирует TensorFlow, — это, как можно понять буквально из названия, *тензор*, или многомерный массив чисел[1].

Переменная в TensorFlow — это некий буфер в памяти, который содержит тензоры. Переменные нужно явным образом инициализировать. Чтобы объявить переменную, нужно задать способ ее инициализации; при желании можно также назначить ей имя, на которое можно будет потом ссылаться. Например:

```
w = tf.Variable(tf.random_normal([3, 2], mean=0.0, stddev=0.4), name='weights')
b = tf.Variable(tf.zeros([2]), name='biases')
```

[1] Читающие это математики наверняка пришли в ужас: как так, тензор — это же элемент тензорного пространства, то есть по сути линейное преобразование между многомерными линейными пространствами. Преобразования тоже образуют линейное пространство, а числа — это всего лишь их координатные представления, они зависят от выбора базиса и могут меняться при том, что сам тензор как геометрический объект останется неизменным. Что ж, вынуждены математиков разочаровать: в TensorFlow тензор — это не настоящий тензор, а именно многомерный массив, никакого мотивированного геометрией отношения эквивалентности на них нет, просто слово красивое. Но раз уж это слово используется насквозь во всей документации и даже в названии TensorFlow, избежать его нам не удастся.

В этом коде мы объявили две переменные: `w` с именем `weights` и `b` с именем `biases`. Для переменных можно явным образом указать, где именно им нужно находиться в памяти; например, если вы хотите объявить переменную на первой (нулевой) видеокарте, это можно сделать так:

```
with tf.device('/gpu:0'):
    w = tf.Variable(tf.random_normal([3, 2], mean=0.0, stddev=0.4), name='weights')
```

Впрочем, в «штатном» режиме, на компьютере с подходящей видеокартой, вам достаточно установить версию TensorFlow с поддержкой GPU, и все тензоры по умолчанию будут инициализироваться на вашей видеокарте. А можно, как мы уже упоминали, и распараллелить все на кластер связанных между собой машин; тогда `tf.device` будет выглядеть примерно так:

```
with tf.device('/job:ps/task:0'):
    w = tf.Variable(tf.random_normal([3, 2], mean=0.0, stddev=0.4), name='weights')
```

Конечно, у инициализации и работы с кластером есть свои тонкости, однако далее в книге примеров с кластерами не будет, поэтому мы не станем подробнее развивать эту тему.

Все переменные обязательно нужно инициализировать. Самый простой способ — сделать это перед началом вычислений:

```
init = tf.initialize_global_variables()
```

Но это не работает в тех случаях, когда нужно инициализировать переменные из значений других переменных; в таких случаях синтаксис будет следующим:

```
w2 = tf.Variable(w.initialized_value(), name='w2')
```

Все переменные текущей сессии можно в любой момент сохранить в файл:

```
saved = saver.save(sess, 'model.ckpt')
```

Эта процедура сохранит все переменные сессии `sess` в файл `model.ckpt`, а чтобы потом восстановить сессию, достаточно запустить

```
saver.restore('model.ckpt')
```

В задачах машинного обучения необходимо повторно применять одну и ту же последовательность операций к разным наборам данных. В частности, обучение с помощью мини-батчей подразумевает периодическое вычисление результата и ошибки на новых примерах. Для удобства передачи новых данных на видеокарту в TensorFlow существуют специальные тензоры — так называемые *заглушки*, `tf.placeholder`, которым нужно сначала передать только тип данных и размерности тензора, а затем сами данные будут подставлены уже в момент вычислений:

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
output = tf.mul(x, y)
with tf.Session() as sess:
    result = sess.run(output, feed_dict={x: 2, y: 3})
```

В `result` после этого должно получиться 6 (проверьте!).

В TensorFlow реализован полный набор операций над тензорами из NumPy с поддержкой матричных вычислений над массивами разной формы и конвертирования между этими формами (broadcasting). Например, в реальных задачах часто возникает необходимость к каждому столбцу матрицы поэлементно добавить один и тот же вектор. В TensorFlow это делается самым простым из возможных способов:

```
m = tf.Variable(tf.random_normal([10, 100], mean=0.0, stddev=0.4), name='matrix')
v = tf.Variable(tf.random_normal([100], mean=0.0, stddev=0.4), name='vector')
result = m + v
```

Здесь `m` — это матрица размера 10×100, а `v` — вектор длины 100, и при сложении `v` будет прибавлен к каждому столбцу `m`. Broadcasting применим для всех поэлементных операций над двумя тензорами и устроен следующим образом. Размерности двух тензоров последовательно сравниваются начиная с конца; при каждом сравнении необходимо выполнение одного из двух условий:

- либо размерности равны;
- либо одна из размерностей равна 1.

При этом тензоры не обязаны иметь одинаковую размерность: недостающие измерения меньшего из тензоров будут интерпретированы как единичные. Размерность получаемого на выходе тензора, если все условия выполнены, вычисляется как максимум из соответствующих размерностей исходных тензоров. Впрочем, это звучит довольно сложно, так что рекомендуем внимательно проверять, как именно будет работать broadcasting в каждом конкретном нетривиальном случае.

Помимо бинарных операций, в TensorFlow реализован широкий ассортимент унарных операций: возведение в квадрат, взятие экспоненты или логарифма, а также широкий спектр редукций. Например, иногда нам бывает необходимо вычислить среднее значение не по всему тензору, а, скажем, по каждому элементу минибатча. Если первую размерность тензора мы интерпретируем как размер минибатча, то соответствующий код может быть таким:

```
tensor = tf.placeholder(tf.float32, [10, 100])
result = tf.reduce_mean(tensor, axis=1)
```

При этом среднее будет вычисляться по второй размерности тензора `tensor`, то есть мы десять раз усредним по 100 чисел и получим вектор длины 10.

В некоторых задачах может возникнуть необходимость использования одних и тех же тензоров переменных для нескольких различных путей вычислений. Предположим, что у нас есть функция, создающая тензор линейного преобразования над вектором:

```
def linear_transform(vec, shape):
    w = tf.Variable(tf.random_normal(shape, mean=0.0, stddev=1.0),
        name='matrix')
    return tf.matmul(vec, w)
```

И мы хотим применить это преобразование к двум разным векторам:

```
result1 = linear_transform(vec1, shape)
result2 = linear_transform(vec2, shape)
```

Понятно, что в этом случае каждая из функций создаст свою собственную матрицу преобразования, что не приведет к желаемому результату. Можно, конечно, задать тензор `w` заранее и передать его в функцию `linear_transform` в качестве одного из аргументов, однако это нарушит принцип инкапсуляции, которым не все готовы пожертвовать. Для подобных случаев в TensorFlow существуют *пространства переменных* (variable scopes). Этот механизм состоит из двух основных функций:

- `tf.get_variable(<name>, <shape>, <initializer>)` создает или возвращает переменную с заданным именем;
- `tf.variable_scope(<scope_name>)` управляет пространствами имен, использующихся в `tf.get_variable()`.

Одним из параметров функции `tf.get_variable()` является инициализатор: вместо того чтобы при объявлении новой переменной в явном виде передавать значения, которыми она должна быть инициализирована, мы можем передать функцию инициализации, которая, получив при необходимости данные о размерности переменной, инициализирует ее. Давайте немного изменим `linear_transform()`:

```
def linear_transform(vec, shape):
    with tf.variable_scope('transform'):
        w = tf.get_variable('matrix', shape,
            initializer=tf.random_normal_initializer())
        return tf.matmul(vec, w)
```

Теперь, если мы попробуем применить наше преобразование к двум векторам последовательно, во время второго вызова увидим ошибку:

```
# Raises ValueError(... transform/matrix already exists ...)
```

Так происходит потому, что `tf.get_variable()` проверяет существование переменной в текущем пространстве имен, чтобы предотвратить связанные с именами ошибки, обычно трудно поддающиеся отладке. Чтобы повторно использовать переменные в пространстве имен, нужно в явном виде сообщить об этом TensorFlow:

```
with tf.variable_scope('linear_transformers') as scope:
    result1 = linear_transform(vec1, shape)
    scope.reuse_variables()
    result2 = linear_transform(vec2, shape)
```

Теперь все работает именно так, как мы хотели с самого начала.

Давайте рассмотрим простейший пример модели, своего рода Hello World для TensorFlow — обучение линейной регрессии. Напомним, что линейная регрессия — это фактически один нейрон, который получает на вход вектор значений $\boldsymbol{x}$, выдает число, и на данных $\{(\boldsymbol{x}_1, y_1),(\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)\}$ задача состоит в том, чтобы минимизировать сумму квадратов отклонений оценок нейрона $\hat{y}_i$ от истинных значений y_i:

$$L = \sum_{i=1}^{N} (\hat{y} - y)^2 \to \min .$$

Но хоть задача и очень простая, мы постараемся в этом примере продемонстрировать полный цикл типичной программы на TensorFlow, с градиентным спуском, обучением на мини-батчах, заглушками и блекджеком.

Вот как это можно сделать. В коде, приведенном ниже, мы будет приближать линейной регрессией функцию вида $f = kx + b$ для $k = 2$ и $b = 1$; k и b будут параметрами, которые мы хотим обучить. Давайте сначала посмотрим на всю программу целиком, а потом подробно разберем, что она делает.

```
import numpy as np,tensorflow as tf
n_samples, batch_size, num_steps = 1000, 100, 20000
X_data = np.random.uniform(1, 10, (n_samples, 1))
y_data = 2 * X_data + 1 + np.random.normal(0, 2, (n_samples, 1))

X = tf.placeholder(tf.float32, shape=(batch_size, 1))
y = tf.placeholder(tf.float32, shape=(batch_size, 1))

with tf.variable_scope('linear-regression'):
    k = tf.Variable(tf.random_normal((1, 1)), name='slope')
    b = tf.Variable(tf.zeros((1,)), name='bias')

y_pred = tf.matmul(X, k) + b
loss = tf.reduce_sum((y - y_pred) ** 2)
optimizer = tf.train.GradientDescentOptimizer().minimize(loss)

display_step = 100
with tf.Session() as sess:
    sess.run(tf.initialize_global_variables())
    for i in range(num_steps):
        indices = np.random.choice(n_samples, batch_size)
        X_batch, y_batch = X_data[indices], y_data[indices]
        _, loss_val, k_val, b_val = sess.run([ optimizer, loss, k, b ],
            feed_dict = { X : X_batch, y : y_batch })
        if (i+1) % display_step == 0:
            print('Эпоха %d: %.8f, k=%.4f, b=%.4f' %
                (i+1, loss_val, k_val, b_val))
```

Теперь давайте посмотрим, что делает эта программа. Мы последовательно:

1) сначала создаем случайным образом входные данные `X_data` и `y_data` по такому алгоритму:
   - набрасываем 1000 случайных точек равномерно на интервале $[0; 1]$;
   - подсчитываем для каждой точки x соответствующий «правильный ответ» y по формуле $y = 2x + 1 + \epsilon$, где ϵ — случайно распределенный шум с дисперсией 2, $\epsilon \sim \mathcal{N}(\epsilon; 0, 2)$;

2) затем объявляем `tf.placeholder` для переменных `X` и `y`; на этом этапе уже нужно задать им размерность, и это в нашем случае матрица размерности (размер мини-батча $\times$ 1) для X и просто вектор длины в размер мини-батча для y;
3) далее инициализируем переменные `k` и `b`; это переменные TensorFlow, которые пока никаких значений не имеют, но будут инициализированы стандартным нормальным распределением для `k` и нулем для `b`;
4) потом мы задаем собственно суть модели и при этом строим функцию ошибки $\sum(\hat{y}-y)^2$; обратите внимание на функцию `reduce_sum`: на выходе она всего лишь подсчитывает сумму матрицы по строчкам, но пользоваться надо именно ею, а не обычной суммой или соответствующими функциями из `numpy`, потому что так TensorFlow сможет куда более эффективно оптимизировать процесс вычислений;
5) вводим переменную `optimizer` — оптимизатор, то есть собственно алгоритм, который будет подсчитывать градиенты и обновлять веса; мы выбрали стандартный оптимизатор стохастического градиентного спуска; сейчас нам важно лишь отметить, что теперь каждый раз, когда мы просим TensorFlow подсчитать значение переменной `optimizer`, где-то за кулисами будут происходить обновления переменных, от которых зависит оптимизируемая переменная `loss`, то есть `k` и `b`; по `X` и `y` оптимизации не будет, потому что значения `tf.placeholder` должны быть жестко заданы, — это входные данные;
6) записываем большой цикл, собственно, делает эти обновления (то есть много раз вычисляет переменную `optimizer`); на каждой итерации цикла мы берем случайное подмножество из `batch_size` (то есть 100) индексов данных и подсчитываем значения нужных переменных; мы подаем в функцию `sess.run` список переменных, которые нужно подсчитать (главное — «вычислить» переменную `optimizer`, остальные нужны только для отладочного вывода), и словарь `feed_dict`, в который записываем значения входных переменных, обозначенных ранее как `tf.placeholder`.

Обратите внимание, что TensorFlow чрезвычайно чувствительна к формам (в смысле shapes, размерностям) тензоров. Например, чтобы работала функция `tf.matmul`, нужно подать ей на вход обязательно матрицы, даже если это матрица размера 1 $\times$ 1, как у нас; а вот складывать результат можно и с обычным вектором длины 1 (да, это разные вещи!), форма которого задается как `(1,)`.

В результате этот код будет выписывать поэтапно уменьшающуюся ошибку и постепенно уточняющиеся значения k и b:

```
Эпоха 100: 5962.15625000, k=0.8925, b=0.0988
Эпоха 200: 5312.11621094, k=0.9862, b=0.1927
Эпоха 300: 3904.57006836, k=1.0761, b=0.2825
...
Эпоха 19900: 429.79974365, k=2.0267, b=0.9006
Эпоха 20000: 378.41503906, k=2.0179, b=0.8902
```

Естественно, точных значений $k = 2$, $b = 1$ на выходе не получится: и градиентный спуск является всего лишь приближенным методом оптимизации, и, что в данном случае важнее, правильный ответ на задачу оптимизации не обязательно совпадает с «задуманными» нами значениями, ведь данные генерировались случайным образом. Отметим, что вместо блока:

```
with tf.Session() as sess
```

можно было бы написать:

```
sess = tf.InteractiveSession()
```

и дальше пользоваться переменной `sess` как постоянно открытой сессией. Это полезно в интерактивном режиме *ipython* или *jupyter notebook*, и в последующих примерах этой книги мы тоже обычно будем пользоваться интерактивными сессиями TensorFlow, чтобы можно было писать и запускать код не подряд, перемежая его комментариями.

Итак, мы рассмотрели простейшие операции с использованием библиотеки TensorFlow и подробно, строчка за строчкой, разобрали первый настоящий пример ее применения. Мы рекомендуем читателю по мере необходимости обращаться к документации TensorFlow [523], да и сами далее еще не раз объясним все соответствующие подробности.

Вторая библиотека, которой мы часто будем пользоваться в этой книге, — это Keras [79]. Она по сути является «оберткой» над библиотеками автоматического дифференцирования; впрочем, в этой книге мы будем рассматривать их как две отдельные сущности[1]. Keras реализует практически все то, о чем мы будем говорить в этой книге, в виде готовых примитивов: можно сказать «сделай мне сверточный слой с такими-то параметрами», и Keras сделает. Однако «за кулисами» происходит все то же самое, что в TensorFlow: строится граф вычислений, который Keras потом скормит другой библиотеке для вычисления градиентов и реализации оптимизационных алгоритмов.

Конечно, в основном Keras хорош именно тем, что в нем много чего уже реализовано в готовом виде. Однако для примера полного цикла на Keras тоже вполне сгодится модель попроще. Давайте реализуем логистическую регрессию, тем более что по сути это как раз и есть «однослойная нейронная сеть».

Сначала импортируем из Keras все, что нам дальше потребуется:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Input, Dense, Activation
```

Затем создадим модель логистической регрессии. Мы делаем следующее:

[1] Однако в официальную первую версию TensorFlow, выход которой состоялся весной 2017 года, библиотека Keras вошла как составная часть. К счастью, совместимость при этом практически не сломалась, и почти любой код, который раньше начинался с import keras, теперь будет нормально работать, начинаясь с import tensorflow.contrib.keras as keras.

1) сначала создаем модель виде `Sequential` — это значит, что модель будет создаваться последовательно слой за слоем;
2) затем добавляем один плотный слой, входы которого будут размерности два (просто для нашего примера), а на выходе будет логистический сигмоид от входов; позже мы в подробностях поговорим о том, как это работает, а сейчас просто поверьте, что это и есть модель логистической регрессии: линейная функция с обучаемыми весами от входов, а затем логистический сигмоид от результата;
3) далее модель собственно компилируется с заданной целевой функцией (мы используем перекрестную энтропию для бинарной классификации) и метрикой точности, которую модель будет подсчитывать и выводить в процессе; в качестве алгоритма оптимизации мы укажем простой стохастический градиентный спуск, до чего-то более сложного мы пока просто не добрались.

Вот как это выглядит в коде:

```
logr = Sequential()
logr.add(Dense(1, input_dim=2, activation='sigmoid'))
logr.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

А теперь можно пробовать на данных. Давайте для этого учебного примера просто сгенерируем два двумерных нормальных распределения, одно с центром в точке $(-1; -1)$, другое в точке $(1; 1)$, с дисперсией 1 по обоим компонентам:

```
def sampler(n, x, y):
    return np.random.normal(size=[n, 2]) + [x, y]

def sample_data(n=1000, p0=(-1., -1.), p1=(1., 1.)):
    zeros, ones = np.zeros((n, 1)), np.ones((n, 1))
    labels = np.vstack([zeros, ones])
    z_sample = sampler(n, x=p0[0], y=p0[1])
    o_sample = sampler(n, x=p1[0], y=p1[1])
    return np.vstack([z_sample, o_sample]), labels

X_train, Y_train = sample_data()
X_test, Y_test = sample_data(100)
```

Таким образом, мы сгенерировали 2000 точек, по тысяче в каждый класс, для обучающего множества и 200 отдельных точек, по 100 в каждый класс, для тестового. Теперь можно запускать обучение; в строчке ниже мы задаем тренировочное множество, число эпох и размер мини-батча:

```
logr.fit(X_train, Y_train, batch_size=16, nb_epoch=100,
    verbose=1, validation_data=(X_test, Y_test))
```

Валидационное множество, которое мы тоже туда передаем, никак не используется при обучении, оно нужно только для того, чтобы считать на нем ту самую метрику `'accuracy'`, которую мы задали выше.

При обучении Keras будет выдавать в удобной форме информацию о текущем состоянии обучения, и выход будет выглядеть примерно так:

```
Train on 2000 samples, validate on 200 samples
Epoch 1/100
2000/2000 [=...=] - 0s - loss: 0.8877 - acc: 0.4400 - val_loss: 0.5625 - val_acc: 0.7450
Epoch 2/100
2000/2000 [=...=] - 0s - loss: 0.4588 - acc: 0.8115 - val_loss: 0.3564 - val_acc: 0.8650
Epoch 3/100
2000/2000 [=...=] - 0s - loss: 0.3402 - acc: 0.8860 - val_loss: 0.2832 - val_acc: 0.9000
Epoch 4/100
2000/2000 [=...=] - 0s - loss: 0.2909 - acc: 0.9080 - val_loss: 0.2463 - val_acc: 0.9150
Epoch 5/100
2000/2000 [=...=] - 0s - loss: 0.2645 - acc: 0.9125 - val_loss: 0.2241 - val_acc: 0.9250
...
Epoch 100/100
2000/2000 [=...=] - 0s - loss: 0.1957 - acc: 0.9255 - val_loss: 0.1399 - val_acc: 0.9350
```

Это значит, что в конечном счете мы обучились классифицировать наши два нормальных распределения на тренировочном множестве с точностью около 92,5 %, а на тестовом — целых 93,5 % (это, конечно, просто случайность). В наших экспериментах алгоритм сходился где-то за 5–10 эпох; «ванильный» стохастический градиентный спуск — не самый лучший алгоритм на свете, так что вполне возможно, что это можно улучшить (и в разделе 4.4 мы поговорим о том, как именно). Пока это только самые первые примеры; мы еще раз подробно пройдемся по тому, как инициализировать и использовать TensorFlow и Keras, в разделе 3.6.

И последняя деталь. В этой книге мы будем приводить примеры в виде кода, который Keras может автоматически компилировать как на TensorFlow, так и на Theano. Более того, как TensorFlow, так и Theano умеют автоматически переводить после этого вычисления на видеокарту. Граф вычислений будет тогда храниться в видеопамяти, а сами вычисления станут гораздо более параллельными и, как следствие, гораздо более эффективными. С точки зрения самого кода, определяющего модель, не меняется абсолютно ничего: чтобы запустить модель на видеокарте, скомпилировав ее в TensorFlow, достаточно просто запустить скрипт в соответствующей версии TensorFlow.

Но есть один небольшой трюк, связанный с выделением памяти. Если вы компилируете модели в Theano, об этом можно не задумываться: Theano сначала компилирует граф целиком и поэтому может сама выделить ровно столько памяти, сколько нужно. А вот TensorFlow[1] сама не справится, по умолчанию она просто съест всю доступную видеопамять, что не позволит нам обучать на одной и той же видеокарте несколько моделей одновременно, даже если бы памяти на них хватило. Поэтому видеопамять для сессии TensorFlow придется выделять вручную; вот как это можно сделать:

[1] По крайней мере, в начале 2017 года. Такие вещи, конечно, меняются очень быстро.

```
def get_session(gpu_fraction=0.2):
    num_threads = os.environ.get('OMP_NUM_THREADS')
    gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=gpu_fraction)

    if num_threads:
        return tf.Session(config=tf.ConfigProto(
            gpu_options=gpu_options, intra_op_parallelism_threads=num_threads))
    else:
        return tf.Session(config=tf.ConfigProto(gpu_options=gpu_options))
```

После этого вы используете эту функцию, чтобы выделить нужную долю видеопамяти:

```
sess = get_session()
```

Вот и все. Если ваша модель помещается в одну пятую видеопамяти, как в строчке выше, значит, вы можете параллельно обучать на своей видеокарте сразу пять таких моделей. Правда, если памяти не хватит, TensorFlow не сможет понять это заранее и выделить больше самостоятельно, а просто «упадет» с ошибкой.

Итак, в этой главе мы пробежались по всем тем предварительным сведениям, которые потребуются нам для обучения нейронных сетей, в том числе самых сложных и глубоких. Пора к ним и переходить, но начнем мы даже не с нейронной сети, а с ее базового компонента: модели одного-единственного нейрона.

Глава 3
Перцептрон,
или Эмбрион мудрого компьютера

TL;DR

В третьей главе мы подробно рассмотрим главную составляющую любой нейронной сети — перцептрон, а также то, как перцептроны соединяются в сети. В частности, мы поговорим:

- об истории искусственных нейронных сетей;
- об определении перцептрона и методах его обучения;
- о разных нелинейных функциях активации, от классических до современных;
- о том, похожа ли наша модель перцептрона на настоящие живые нейроны;
- о том, как соединять нейроны в сеть и почему это совсем не такое простое дело, как могло бы показаться.

А затем, обсудив все это, приведем живой пример сети, которая обучится распознавать рукописные цифры.

3.1. Когда появились искусственные нейронные сети

> Но и простой гражданин должен читать Историю. Она мирит его с несовершенством видимого порядка вещей, как с обыкновенным явлением во всех веках...
>
> *Н. М. Карамзин.*
> *История государства Российского*

Как мы уже говорили, нейронные сети — это пример математической конструкции, мотивированной и вдохновленной исследованиями человеческого мозга. Поэтому довольно естественно, что по меркам истории математики нейронные сети — достаточно молодой объект; они, очевидно, не могли появиться во времена Аристотеля, который считал, что мозг охлаждает кровь, и люди тем и отличаются от животных, что имеют большой орган для охлаждения крови и потому могут действовать более рационально. Однако по меркам истории искусственного интеллекта нейронные сети — это одна из старейших, самых первых конструкций, появившаяся еще до знаменитого эссе Тьюринга *Computing Machinery and Intelligence*, до Дартмутского семинара, до того, как появился собственно термин «искусственный интеллект».

По видимому, первой работой, предлагающей математическую модель нейрона и конструкцию искусственной нейронной сети, была статья Уоррена Маккаллоха (Warren McCulloch) и Уолтера Питтса[1] [356], опубликованная в 1943 году. Авторы отмечают, что из-за бинарной природы нейронной активности (нейрон либо «включен», либо «выключен», практически без промежуточных состояний) нейроны удобно описывать в терминах пропозициональной логики, а для нейронных сетей разрабатывают целый логический аппарат, формализующий ациклические графы. Сама конструкция искусственного нейрона, который у Маккаллоха и Питтса называется Threshold Logic Unit (TLU), или Linear Threshold Unit, получилась очень современной: линейная комбинация входов, которая затем поступает на вход нелинейности в виде «ступеньки», сравнивающей результат с некоторым порогом (threshold).

С одной стороны, работа Маккаллоха и Питтса еще не относилась к машинному обучению: в ней модели нейрона и нейронной сети были введены чисто логически, как система аксиом и правил вывода; затем с помощью этих правил был доказан ряд общих теорем. Авторы скорее рассуждали о том, что вообще можно было бы

[1] *Уолтер Питтс* (Walter Pitts, 1923–1969) — американский логик, специализировавшийся на вычислительной нейробиологии. В детстве Уолтер сам обучился логике и математике; когда ему было 12 лет, Питтс три дня подряд сидел в библиотеке и читал *Principia Mathematica*, а потом написал Бертрану Расселу критическое письмо, в котором выделил несколько серьезных проблем с первым томом книги. Время было совсем другое, когнитивной перегрузки меньше, а спам-фильтров не существовало вовсе, и Рассел письмо не только прочитал, но принял всерьез и пригласил Уолтера к себе в Кембридж; Питтс не поехал, однако стал заниматься логикой и продолжал переписку с Расселом, пока учился в университете Чикаго. О том, что было дальше, читайте в основном тексте главы.

сделать с помощью таких искусственных нейронов, основанных на сравнении с порогом, чем пытались предложить конкретные работающие алгоритмы для этого. Такая «нейронная сеть» еще не умела обучаться ни в каком смысле слова, и ее непосредственный эффект был скорее в том, чтобы вообще предложить идею формализации нейронных сетей и нейронной активности, показав, что у нас в голове вполне может содержаться собранная из нейронов машина Тьюринга (идея машины Тьюринга тогда тоже была совсем свежей).

Статья Маккаллоха и Питтса прошла практически незамеченной среди нейробиологов, но создатель кибернетики Норберт Винер сразу понял перспективы искусственных нейронных сетей и идеи о том, как мышление может самопроизвольно возникать из таких простых логических элементов. Винер познакомил Маккаллоха и Питтса с фон Нейманом, и они впятером, вместе с когнитивистом Джеромом Леттвином, стали работать над тем, как адаптировать статистическую механику для моделирования мышления, а потом и построить работающий компьютер на основе моделирования нейронов. Память в таком компьютере предполагалось получить из замкнутых контуров активации нейронов, когда последовательная активация превращается в самоподдерживающийся процесс (в настоящем мозге очень много таких замкнутых контуров, и то, как они работают и для чего нужны, до сих пор не вполне ясно). Питтс считался самым гениальным ученым в этой группе и объявил, что пишет диссертацию о вероятностных трехмерных нейронных сетях, с трехмерной структурой связей между ними.

Диссертация Питтса наверняка стала бы очередным прорывом в кибернетике, но, к сожалению, закончилось все трагически. Жене Винера Маргарет категорически не нравились вечеринки, которые устраивал Маккаллох на своей ферме, и когда в 1952 году Маккаллох объявил, что переезжает в Кембридж, Маргарет рассказала Винеру о том, что эти «мальчики» якобы соблазнили их дочь Барбару, пока та гостила в доме Маккаллоха в Чикаго. На деле ничего такого, конечно, не было, но Винер поверил и мгновенно прекратил всякое общение с Маккаллохом и Питтсом. Для Питтса, который к тому времени уже был подвержен депрессиям, это стало поворотным моментом: он стал все больше пить, перестал появляться в MIT, сжег (!) свою диссертацию и все записи о ней, полностью прекратил заниматься наукой и умер в 46 лет от кровоизлияния, связанного с циррозом печени [174].

Другой первый шаг искусственных нейронных сетей, также весьма релевантный современным исследованиям, — это книга Дональда Хебба *The Organization of Behaviour*, вышедшая в 1949 году [209]. Сама книга относится скорее к нейробиологии, чем к математике, но некоторые ее части содержат ключевые идеи, послужившие основой всего дальнейшего обучения нейронных сетей. Основная конкретная идея, перешедшая из работ Хебба в современное машинное обучение практически без изменений, — это так называемое *правило Хебба*, которое сам Дональд Хебб в первоисточнике формулировал так: «Когда аксон клетки А находится достаточно близко, чтобы возбудить клетку В, и многократно или постоянно участвует в том, чтобы ее активировать, в одной или обеих клетках происходит некий процесс роста или изменение метаболизма, в результате которого эффективность А как клетки,

возбуждающей B, увеличивается». Проще говоря, если связь между двумя нейронами часто используется, она от этого упражняется и становится сильнее. Эта простая, но очень мощная идея не только мотивировала дальнейшие исследования, но и сама по себе легла в основу так называемого *обучения по Хеббу* (Hebbian learning), группы методов обучения без учителя, основанных на этом базовом правиле. В данной книге мы не будем подробно останавливаться на обучении по Хеббу, а просто упомянем сети Хопфилда (Hopfield networks) [230], обучение которых основано на этом принципе, а также недавние работы, связанные со временем активации нейронов. Дело в том, что Хебб говорил о *причинном* участии клетки A в активации нейрона B, а не просто одновременном срабатывании, то есть клетка A должна была все же срабатывать чуть раньше; современное развитие этой идеи известно как пластичность, зависящая от времени спайков (spike-timing dependent plasticity) [351, 497].

Как только идеи Хебба оформились, тут же последовала их программная реализация; точнее, в те времена это была, конечно, реализация «в железе». В 1951 году Марвин Минский, которого мы уже не раз упоминали, и его аспирант Дин Эдмунд (Dean Edmund) построили сеть из сорока синапсов. Синапсы были случайным образом соединены друг с другом и обучались по правилам Хебба на основе вознаграждений, которые им давали исследователи. Эта модель получила название SNARC (Stochastic Neural Analog Reinforcement Calculator), и хотя непосредственного развития она не получила, можно сказать, что это первая настоящая реализация обучения с подкреплением (о котором речь пойдет в главе 9).

Следующим прорывом в нейробиологии стали работы Хьюбела и Визеля [235, 236, 569], которые смогли достаточно подробно изучить активации нейронов в зрительной коре, что стало мотивацией для появления сверточных нейронных сетей и в некотором смысле глубокого обучения вообще (см. раздел 5.1). Но это было уже после появления первых перцептронов Розенблатта [455, 456], о которых пойдет речь в следующем разделе.

Завершая этот краткий исторический экскурс (впрочем, вся эта глава во многом рассказывает именно об истории развития нейронных сетей, так что к ней мы еще не раз вернемся), упомянем один подробный обзор современной истории нейронных сетей, работу Юргена Шмидхубера[1] [475]. Несмотря на то что это очень плотный и довольно сухой текст, зачастую представляющий собой просто набор ссылок, он выгодно отличается от многих других обзоров и исторических очерков тем, что анализирует именно историю *идей*: как развивались не просто нейронные

[1] *Юрген Шмидхубер* (Jurgen Schmidhuber, p. 1963) — немецкий математик и информатик, один из отцов-основателей современного машинного обучения, знаменитый своими достижениями в области нейронных сетей, генетических алгоритмов, теории сложности и других областях. В частности, лаборатория Шмидхубера разработала ряд конструкций современных рекуррентных нейронных сетей, включая LSTM (long short-term memory). Одно любопытное направление исследований Шмидхубера — теория красоты, основанная на понятии колмогоровской сложности, и минималистические произведения искусства, создаваемые очень простыми алгоритмами (low complexity art) [473, 474].

сети в целом, а конкретные конструкции и алгоритмы оптимизации, чего они достигали на каждом этапе развития, и где сейчас все эти идеи применяются. Кроме того, автор старается проследить историю каждой идеи до самых ее истоков, до первого появления в литературе, зачастую с неожиданными результатами. В подготовке этой книги мы не раз использовали обзор Шмидхубера и искренне его рекомендуем.

3.2. Как работает перцептрон

> К 80-м годам двадцатого столетия... Минский и Гуд разработали методику автоматического зарождения и самовоспроизведения нервных цепей в соответствии с любой произвольно выбранной программой. Оказалось, что искусственный мозг можно «выращивать» посредством процесса, поразительно сходного с развитием человеческого мозга. Точные детали этого процесса в каждом отдельном случае так и оставались неизвестными; впрочем, будь они даже известны, человеческий разум не смог бы постичь всю их сложность.
>
> *А. Кларк. Космическая Одиссея 2001, пер. Н. Галь*

> Мы, люди старого века, мы полагаем, что без принсипов (Павел Петрович выговаривал это слово мягко, на французский манер, Аркадий, напротив, произносил «прынцип», налегая на первый слог), без принсипов, принятых, как ты говоришь, на веру, шагу ступить, дохнуть нельзя.
>
> *И. С. Тургенев. Отцы и дети*

Итак, мы наконец-то приступаем к рассмотрению собственно математики происходящего. И начнем мы, конечно, с азов, с первой конструкции *линейного перцептрона*[1], описанного еще Фрэнком Розенблаттом[2] в 1950-х годах [455, 456].

По сути своей перцептрон Розенблатта — это линейная модель классификации. В главе 1.3 мы уже обсуждали задачи классификации, а здесь будем иметь дело с самой простой, *бинарной* классификацией, когда все объекты в тренировочной выборке помечены одной из двух меток (скажем, $+1$ или -1), и задача состоит

[1] По-русски иногда еще пишут и говорят «персептрон»; в этом споре между Аркадием и Павлом Петровичем мы, пожалуй, займем сторону молодого поколения.

[2] *Фрэнк Розенблатт* (Frank Rosenblatt, 1928–1971) — американский психолог и информатик. Интересно, что Розенблатт был по основной специальности психологом, с упором, конечно, на когнитивные науки. Например, одно из направлений его деятельности после перцептрона — исследования того, можно ли передать память или выученное поведение путем непосредственного физического впрыскивания вещества мозга обученных крыс в необученных (Розенблатт убедительно показал, что нет, нельзя). А еще он интересовался политикой и астрономией, построил домашнюю обсерваторию и активно участвовал в SETI.

в том, чтобы научиться расставлять эти метки и у новых, ранее не виденных объектов. А «линейная модель» означает, что в результате обучения модель разделит все пространство входов на две части гиперплоскостью: правило принятия решения о том, какую метку ставить, будет линейной функцией от входящих признаков.

Для простоты мы будем считать, что каждый вход представляет собой вектор вещественных чисел $\boldsymbol{x} = (x_1,x_2,\ldots,x_d) \in \mathbb{R}^d$, и входы в тренировочном множестве снабжены известными выходами $y(\boldsymbol{x}) \in \{-1,1\}$. Вообще говоря, природа объектов на входе модели — это обычно больной вопрос в машинном обучении: часто нелегко сделать так, чтобы одна и та же модель могла принимать на вход и непрерывные, и дискретные признаки, но мы пока абстрагируемся от всех этих проблем. Тогда в наших терминах «линейная модель» означает, что мы будем искать такие веса $w_0,w_1,\ldots,w_d \in \mathbb{R}^d$, чтобы знак линейной функции

$$\operatorname{sign}\left(w_0 + w_1x_1 + w_2x_2 \ldots + w_dx_d\right)$$

как можно чаще совпадал бы с правильным ответом $y(\boldsymbol{x})$. Для удобства, чтобы не тащить везде за собой свободный член w_0, мы введем в вектор $\boldsymbol{x}$ лишнюю «виртуальную» размерность и будем считать, что $\boldsymbol{x}$ выглядит как $\boldsymbol{x} = (1,x_1,x_2,\ldots,x_d)$, и $\boldsymbol{x} \in \mathbb{R}^{d+1}$; тогда $w_0 + w_1x_1 + w_2x_2 \ldots + w_dx_d$ можно считать просто скалярным произведением $\boldsymbol{w}^\top\boldsymbol{x}$ вектора весов $\boldsymbol{w} = (w_0,w_1,w_2,\ldots,w_d)$ на входной вектор $\boldsymbol{x}$. Естественно, ни задача, ни ответ на нее от этого преобразования не меняются; это стандартный трюк, и мы часто будем им пользоваться.

Как обучать такую функцию? Сначала нужно выбрать функцию ошибки. Было бы, конечно, хорошо выбрать ее как число неверно классифицированных примеров. Но тогда, как мы обсуждали в разделе 2.3, получится кусочно-гладкая функция ошибки с массой разрывов: она будет принимать только целые значения и резко менять их при переходе от одного числа неверно классифицированных примеров к другому. Градиентный спуск к такой функции не применишь, и становится совершенно непонятно, как обучать $\boldsymbol{w}$. Поэтому в перцептроне Розенблатта используется другая функция ошибки, так называемый *критерий перцептрона*:

$$E_P(\boldsymbol{w}) = -\sum_{\boldsymbol{x}\in\mathcal{M}} y(\boldsymbol{x})\left(\boldsymbol{w}^\top\boldsymbol{x}\right),$$

где $\mathcal{M}$ обозначает множество тех примеров, которые перцептрон с весами $\boldsymbol{w}$ классифицирует неверно.

Иначе говоря, мы минимизируем суммарное отклонение наших ответов от правильных, но только в неправильную сторону; верный ответ ничего не вносит в функцию ошибки. Умножение на $y(\boldsymbol{x})$ здесь нужно для того, чтобы знак произведения всегда получался отрицательным: если правильный ответ -1, значит, перцептрон выдал положительное число (иначе бы ответ был верным), и наоборот. В результате у нас получилась кусочно-линейная функция, дифференцируемая почти везде, а этого вполне достаточно.

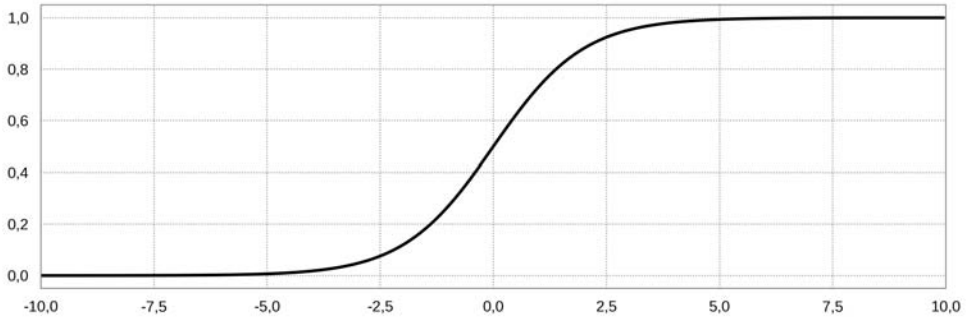


Рис. 3.1. Логистический сигмоид

Теперь мы можем оптимизировать ее градиентным спуском. На очередном шаге получаем:

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta\nabla_{\boldsymbol{w}} E_P(\boldsymbol{w}) = \boldsymbol{w}^{(\tau)} + \eta t_n \boldsymbol{x}_n.$$

Алгоритм такой — мы последовательно проходим примеры $\boldsymbol{x}_1$,$\boldsymbol{x}_2, \ldots$ из обучающего множества, и для каждого $\boldsymbol{x}_n$:

- если он классифицирован правильно, не меняем ничего;
- а если неправильно, прибавляем $\eta t_n \boldsymbol{x}_n$ к $\boldsymbol{w}$.

Ошибка на примере $\boldsymbol{x}_n$ при этом, очевидно, уменьшается, но, конечно, совершенно никто не гарантирует, что вместе с тем не увеличится ошибка от других примеров. Это правило обновления весов так и называется — *правило обучения перцептрона*, и это было основной математической идеей работы Розенблатта.

Но и это еще не все. Чтобы двигаться дальше, нам нужно добавить в перцептрон еще один компонент — так называемую *функцию активации*. Дело в том, что в реальности перцептроны, как мы увидим буквально в следующем разделе, не могут быть линейными, как мы их определили сейчас: если они останутся линейными, то из них невозможно будет составить содержательную сеть. На выходе перцептрона обязательно присутствует нелинейная *функция активации*, которая принимает на вход все ту же линейную комбинацию. Функции активации бывают разными, и мы их подробно рассмотрим в разделе 3.3. Но самая классическая, наиболее популярная исторически и до сих пор часто используемая функция активации — это *логистический сигмоид*:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

График его показан на рис. 3.1. Как и другие функции активации нейронов, это монотонно неубывающая функция, которая при $x \to -\infty$ стремится к нулю, а при $x \to \infty$ стремится к единице; неформально говоря, это значит, что если на вход подают большое отрицательное число, то нейрон совсем не активируется, а если

большое положительное, то активируется почти наверняка. Эта функция называется сигмоидом как раз потому, что форма ее похожа на букву S[1].

Обучать один такой перцептрон несложно: можно применить все тот же градиентный спуск. Разница только в том, что теперь мы рассматриваем задачу бинарной классификации, и данные $y(\boldsymbol{x})$ представляют собой метки 0 и 1, а функция ошибки выглядит как перекрестная энтропия (вспомним раздел 2.3):

$$E(\boldsymbol{w}) = -\frac{1}{N}\sum_{i=1}^{N}\Big(y_i \log \sigma(\boldsymbol{w}^\top \boldsymbol{x}_i) + (1-y_i)\log\Big(1-\sigma(\boldsymbol{w}^\top \boldsymbol{x}_i)\Big)\Big).$$

От этой функции по-прежнему несложно взять производную. В итоге перцептрон с логистическим сигмоидом в качестве функции активации фактически реализует логистическую регрессию и строит при этом линейные разделяющие поверхности.

Еще стоит отметить, что один перцептрон — это просто разновидность линейной регрессии; мы говорили о линейной регрессии в разделе 2.2, где функцией ошибки был квадрат отклонения. И разные нелинейности на выходе одного нейрона можно рассматривать как разные формы функции ошибки. В обычной линейной регрессии нелинейности нет совсем, и ошибка считается как сумма квадратов отклонений: $E(\boldsymbol{x}, y) = (y - \boldsymbol{w}^\top \boldsymbol{x})^2$. В логистической регрессии добавляется сигмоид, и ошибка теперь считается как перекрестная энтропия; от этого задача регрессии превращается в задачу классификации, а выходы нейрона теперь можно интерпретировать как вероятности. И другие нелинейности, о которых мы поговорим ниже, тоже можно рассматривать как разные формы функции ошибки для базовой конструкции перцептрона — линейной комбинации входов.

Теперь, когда мы разобрались с конструкцией одного перцептрона, мы можем собрать из них целую сеть. Она будет представлять собой граф вычислений, который мы рассматривали в разделе 2.5. Один перцептрон может служить одним узлом в этом графе, выступая в роли элементарной функции: нам для этого требуется только уметь считать частные производные по всем переменным, что для перцептрона сделать совсем не сложно. В этой паре предложений произошел гигантский скачок: теперь мы фактически умеем обучать любые нейронные сети (кроме пока что рекуррентных), и для этого годится общий алгоритм обратного распространения, который мы обсудили в разделе 2.5. Конечно, обучать сложные сети буквально стохастическим градиентным спуском — не самая лучшая идея. Вся глава 4 будет посвящена разным способам заставить обучение работать лучше. И все же уже сейчас мы теоретически можем обучить сколь угодно сложную нейронную сеть!

---

[1] Это, конечно, только если очень хорошенько присмотреться и как следует прищуриться, однако понятие «S-образной формы» действительно довольно широкое и включает в том числе и такие картинки. Вспомните, например, что значок интеграла тоже происходит от буквы S, от слова «сумма». Правда, Лейбниц в знаке интеграла использовал не базовую латинскую S, а «долгую S»; но она, в свою очередь, происходит от прописной латинской s в курсивном написании.

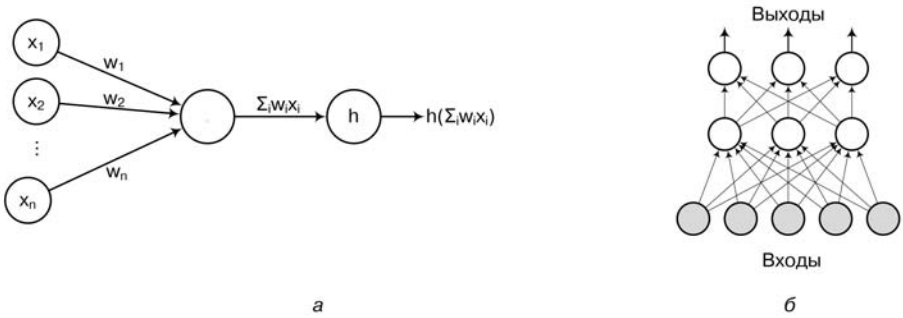


Рис. 3.2. Основная идея архитектуры нейронных сетей: *а* — граф вычислений для перцептрона; *б* — полносвязная нейронная сеть с одним скрытым уровнем и одним выходным уровнем

На рис. 3.2 мы показали структуру одного перцептрона (рис. 3.2, *а*), которую уже подробно обсудили выше, а также пример простой нейронной сети (рис. 3.2, *б*). Каждый вход этой сети подается на вход каждому перцептрону «первого уровня». Затем выходы перцептронов «первого уровня» подаются на вход перцептронам «второго уровня», а их выходы уже считаются выходами всей сети целиком.

Есть еще одно важное замечание по поводу того, как объединять нейроны в сеть. Обычно в любой сети отдельные нейроны объединены в *слои*. Вектор входов подается сразу в несколько параллельных нейронов, у каждого из которых свои собственные веса, а затем выходы этих параллельных нейронов опять рассматриваются как единое целое, новый вектор выходов. Так, на рис. 3.2, *б* изображена сеть с одним скрытым слоем и одним выходным слоем, то есть всего их здесь два.

Казалось бы, для графа вычислений теоретически нет разницы, какую именно архитектуру выбрать, все равно это всего лишь набор независимых нейронов. Однако концепция слоя очень важна с вычислительной, практической точки зрения. Дело в том, что вычисления в целом слое нейронов можно *векторизовать*, то есть представить в виде умножения матрицы на вектор и применения вектор-функции активации с одинаковыми компонентами. Если в слое k нейронов, и веса у них $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_k$, $\boldsymbol{w}_i = \begin{pmatrix} w_{i1} & \cdots & w_{in} \end{pmatrix}^\top$, а на вход подается вектор $\boldsymbol{x} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}^\top$, то в результате мы получим у нейрона с весами $\boldsymbol{w}_i$ выход $y_i = f(\boldsymbol{w}_i^\top \boldsymbol{x})$, где f — его функция активации. Тогда вычисление, которое делают все нейроны сразу, можно будет представить в векторной форме так:

$$\begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix} = \boldsymbol{y} = f(W\boldsymbol{x}) = \begin{pmatrix} f\left(\boldsymbol{w}_1^\top \boldsymbol{x}\right) \\ \vdots \\ f\left(\boldsymbol{w}_k^\top \boldsymbol{x}\right) \end{pmatrix}, \quad \text{где} \quad W = \begin{pmatrix} \boldsymbol{w}_1 \\ \vdots \\ \boldsymbol{w}_k \end{pmatrix} = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & & \vdots \\ w_{k1} & \cdots & w_{kn} \end{pmatrix},$$

и вычисление всего слоя сведется к умножению матрицы весов на вектор входов, а затем покомпонентному применению одной и той же функции активации. При этом оказывается, что матричные вычисления можно реализовать гораздо эффективнее, в частности на графических процессорах (видеокартах), чем те же вычисления, но представленные в виде обычных циклов. Поэтому такое векторизованное представление — это один из главных инструментов для того, чтобы перенести обучение и применение нейронных сетей на видеокарты, а это ускоряет все процессы буквально в десятки раз.

В заключение этого раздела давайте еще на минутку вернемся к истории; теперь мы можем продолжить сюжет, начатый в разделе 1.2, и проследить историю первых перцептронов до конца. История искусственного интеллекта полна, простите за клише, взлетов и падений. Циклы чрезмерного оптимизма и неизбежных после этого разочарований, то, что по-английски называется boom-and-bust cycles, в истории развития методов машинного обучения были особенно яркими, ярче, чем в других науках. И неудивительно, ведь искусственный интеллект часто обещал продвижения, которые, с одной стороны, выглядят совершенно потрясающе — разговаривающие роботы! самодвижущиеся машины! автоматический поиск по всем книгам мира! — а с другой стороны, все время кажутся как бы «сразу за горизонтом», как будто вот-вот, еще немножко, и все получится, андроиды будут приятным голосом сообщать послезавтрашнюю погоду, приносить кофе и писать стихи, а нам останется только чувствовать себя белыми сахибами среди кремниевых слуг.

Конечно, раскручивали очередной виток спирали зачастую не сами ученые, а журналисты, и делали это с фантазией, но надо сказать, что ученые тоже скромностью не отличались. Например, мы с вами уже подробно рассмотрели перцептрон Розенблатта и увидели, что эта модель обучает простой линейный классификатор. А вот что писала о перцептроне *The New York Times* (вовсе не таблоид, так что вряд ли эта косвенная речь сильно искажена) 8 июля 1958 года: «Психолог показывает эмбрион компьютера, разработанного, чтобы читать и становиться мудрее. Разработанный ВМФ... стоивший 2 миллиона долларов компьютер "704", обучился различать левое и правое после пятидесяти попыток... По утверждению ВМФ, они используют этот принцип, чтобы построить первую мыслящую машину класса "Перцептрон", которая сможет читать и писать; разработку планируется завершить через год, с общей стоимостью $100 000... Ученые предсказывают, что позже Перцептроны смогут распознавать людей и называть их по имени, мгновенно переводить устную и письменную речь с одного языка на другой. Мистер Розенблатт сказал, что в принципе возможно построить "мозги", которые смогут воспроизводить самих себя на конвейере и которые будут осознавать свое собственное существование». А в разделе 1.2 мы уже цитировали грантозаявку отцов-основателей на проведение Дартмутского семинара летом 1956 года, в которой они обещали «обучить машины использовать естественные языки, формировать абстракции и концепции... и улучшать самих себя». Оптимистическое было время, что и говорить.

Кстати, именно со «мгновенным переводом устной и письменной речи» вышел казус, который в свое время послужил спусковым крючком для первой настоящей

«зимы искусственного интеллекта». Любопытно, что без русских здесь не обошлось; скорее, правда, Ruskies, чем Russians. Во времена холодной войны американскому правительству очень хотелось получить машину, которая могла бы быстро и надежно переводить документы с русского на английский и обратно. Начиная с 1954 года, соответствующие исследования активно спонсировались, и оптимизм был, опять же, безграничен: начало исследований в искусственном интеллекте совпало со знаменитыми разработками Ноама Хомского[1] о трансформациях и порождающих грамматиках. Казалось, что естественный язык вот-вот получится описать достаточно просто и аналитически... но нет. После десяти лет исследований выяснилось, что машинный перевод — это все-таки очень и очень непростая задача, и даже различать омонимы совсем нелегко. Именно из этих исследований появился знаменитый пример двойного перевода, после которого *the spirit is strong but the flesh is weak* («дух силен, плоть слаба») превращается в *the vodka is good but the meat is rotten* («водка хорошая, но мясо протухло»); и, главное, все правильно! Теоретически такой смысл здесь тоже мог бы быть, просто для нас, знающих естественный язык и его не слишком очевидные особенности, вероятность такого прочтения ничтожно мала. А в середине 1960-х годов этот пример смешным вовсе не показался, ALPAC (Automatic Language Processing Advisory Committee — тогда в США очень любили создавать всевозможные комитеты) в своем отчете заключил, что машинный перевод получается гораздо хуже, дороже и медленнее человеческого, все финансирование свернули, и усилия в направлении машинного перевода прекратились надолго.

За перцептронами пришли немногим позже. В 1969 году Марвин Минский[2] и Сеймур Пейперт опубликовали книгу с нехитрым названием «Перцептроны» [365], которая вызвала серьезное разочарование в конструкции перцептронов

---

[1] *Ноам Хомский* (Noam Chomsky, p. 1928) — американский лингвист, философ, логик и политический активист, «отец современной лингвистики». Основная научная заслуга Хомского — революция в лингвистике, которую он произвел своими работами об универсальных и порождающих грамматиках; можно сказать, что именно благодаря Хомскому математика пришла в лингвистику. Хомский всегда отличался активной гражданской позицией, участвовал в политических дебатах, был не только ученым, но и публичной фигурой, и в наше время Хомский широко известен еще и как политический активист, публицист и критик; он стоит на позициях анархо-синдикализма и либертарианства и последовательно критикует «американский истеблишмент». Кстати, генетически Ноам Хомский — вполне российский ученый, буквально во втором поколении: его отец был украинским евреем, эмигрировавшим в Соединенные Штаты во время Первой мировой войны, а мать приехала из Беларуси.

[2] *Марвин Ли Минский* (Marvin Lee Minsky, 1927–2016) — американский информатик, один из отцов-основателей искусственного интеллекта как области науки. Его знаменитые статьи *Steps Towards Artificial Intelligence* [363] и *Matter, Mind, and Models* [364] содержали постановки важнейших задач на пути к построению мыслящих машин и сыграли центральную роль в развитии искусственного интеллекта и когнитивистики. Кроме заслуг в искусственном интеллекте, он также разработал первый в мире шлем виртуальной реальности (в 1963 году!), а также для нужд биологии изобрел конфокальный микроскоп. Примечательно, что Минский консультировал Артура Кларка при написании и постановке его «Космической Одиссеи 2001» и даже весьма лестно там упоминается — см. эпиграф к разделу 3.2; цитата, кстати, весьма характерная для тех оптимистических времен. К сожалению, в январе 2016 года Марвин Минский скончался; однако он верил или хотя бы недостаточно скептически относился к крионике, и вполне вероятно, что его тело и, главное, мозг сейчас успешно заморожены компанией Alcor.

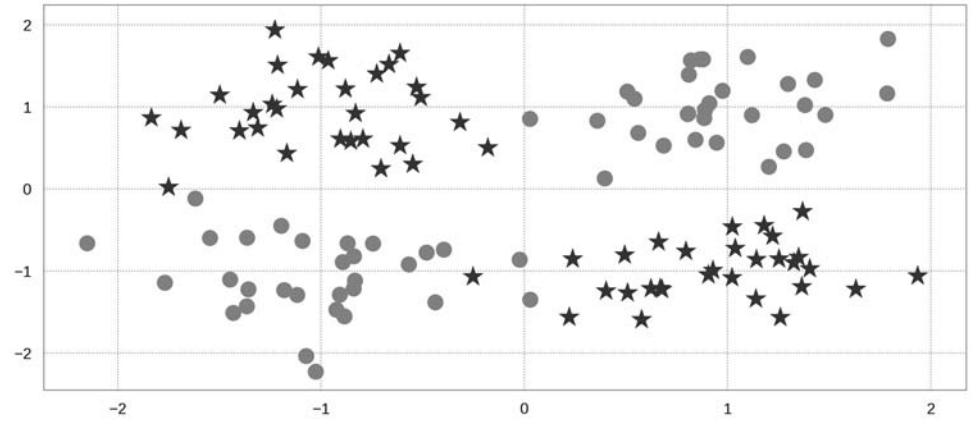


Рис. 3.3. Пример линейно неразделимых множеств

Розенблатта. Эту историю часто рассказывают так, как будто бы основным аргументом Минского и Пейперта была линейность перцептрона. И действительно, очевидно, что один перцептрон Розенблатта может обучиться разделять только те множества точек, между которыми можно провести гиперплоскость (такие множества логично называются *линейно разделимыми*), а на свете есть и масса других множеств! Например, одинокий линейный перцептрон никогда не обучится реализовывать функцию XOR: множество ее нулей и множество ее единиц, увы, линейно неразделимы. Например, на рис. 3.3 мы изобразили два множества точек, которые похожи на значения функции XOR: одно из них (звездочки) порождено из смеси нормальных распределений с центрами в точках $(-1, 1)$ и $(1, -1)$, а другое (точки) — из смеси нормальных распределений с центрами в точках $(-1, -1)$ и $(1, 1)$. Действительно, хотя точки и звездочки очевидно занимают разные области на плоскости и легко отличимы, столь же очевидно, что никакая прямая линия не может адекватно их разделить. Другое возражение против линейной конструкции состоит в том, что комбинировать линейные перцептроны в сеть тоже не имеет никакого смысла: композиция линейных функций снова будет линейной, и сеть из любого, сколь угодно большого числа линейных перцептронов сможет реализовать только те же самые линейные функции, для которых было бы достаточно и одного.

Сегодня нам странно слышать, что это серьезные аргументы против: ну конечно, линейный классификатор не может реализовать XOR, но сеть из нескольких классификаторов с любой нелинейностью справится с этим без труда. Ну конечно, соединять линейные перцептроны в сеть бессмысленно, но как только мы добавим нелинейную функцию активации, даже самую простую, смысл тут же появится, и весьма, простите за каламбур, глубокий. И действительно, это понимали еще Маккаллох и Питтс (они и вовсе предлагали строить аналог машины Тьюринга на

своих перцептронах), и для Минского это тоже, конечно, не было секретом. Негативные утверждения в книге «Перцептроны» касались только некоторых конкретных архитектур: например, Минский и Пейперт показали, что сети с одним скрытым слоем не могут вычислять некоторые функции, если перцептроны скрытого слоя не связаны со всеми входами (раньше люди надеялись, что удастся обойтись «локальными» связями между перцептронами); в целом, ничего криминального. Однако в конце 1960-х годов о нейронных сетях и тем более глубоких сетях еще не было широко известно, хотя разрабатываться они уже начинали; и вышло так, что книга Минского и Пейперта, получившая широкую известность, надолго оттолкнула многих исследователей от того, чтобы продолжать изучение перцептронов и вообще нейронных сетей. Целых десять лет после этого заниматься нейронными сетями было немодно и неприбыльно; грантов на это практически не давали. Тем не менее, в 1970-е годы был достигнут очень серьезный прогресс в разработке и изучении нейронных сетей. Об этом мы поговорим в следующих главах, а пока вернемся к перцептрону и посмотрим, какие у него бывают функции активации в наше просвещенное время.

3.3. Современные перцептроны: функции активации

> В области реального, физического наслаждения человек имеет не больше животного, помимо того, насколько его более потенцированная (возвышенная, утонченная) нервная система усиливает ощущения всякого наслаждения, а также и всякого страдания. Но зато какою силою отличаются возбуждаемые в нем аффекты, сравнительно с ощущениями животного! как несоразмерно сильнее и глубже волнуется его дух! и все из-за того, чтобы напоследок добиться того же результата: здоровья, пищи, крова и т. п.
>
> *А. Шопенгауэр. Parerga und Paralipomena*

В наше время от базовой конструкции перцептрона, конечно же, никто не отказывается. Смысл и основная последовательность по-прежнему те же самые: сначала в перцептрон поступают входы из данных или предыдущих уровней сети, затем берется их линейная комбинация с некоторыми весами, которые, собственно, и будут обучаться в сети, а потом результат проходит через некоторую нелинейную функцию, без которой, как мы уже видели в этой главе, никакой выразительной силы у нейронной сети не получится. Именно из таких перцептронов, или нейронов, состоят все современные нейронные сети. Разница есть только в том, какова, собственно, конструкция нелинейности; и вот на этом вопросе уже стоит остановиться подробнее, он представляется чрезвычайно интересным.

Как мы уже говорили, исторически в нелинейных перцептронах обычно применялась функция активации (возбуждения нейрона) в виде логистического сигмоида: $\sigma(x) = \frac{1}{1+e^{-x}}$. Эта функция обладает всеми свойствами, необходимыми для нелинейности в нейронной сети: она ограничена, стремится к нулю при $x \to -\infty$ и к единице при $x \to \infty$, везде дифференцируема, и производную ее легко подсчитать как $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Но она такая, конечно же, не одна. Есть много разных функций активации, которые в разное время и для разных целей использовались в литературе. Некоторые из них показаны, для наглядности на одном и том же графике, на рис. 3.4; в этом разделе мы с ними познакомимся поближе.

Гиперболический тангенс:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

очень похож по свойствам на логистический сигмоид: он тоже непрерывен, тоже ограничен (правда, он стремится к -1 при $x \to -\infty$, а не к нулю, но это, конечно, не важно), и производную от него тоже легко подсчитать через него самого:

$$\tanh'(x) = 1 - \tanh^2(x)$$

(проверьте сами!). По сравнению с логистическим сигмоидом гиперболический тангенс значительно «круче» растет и убывает, быстрее приближается к своим пределам; например, наклон касательной в нуле у тангенса $\tanh'(0) = 1$, а у логистического сигмоида $\sigma'(0) = \frac{1}{4}$.

Однако есть и более важная тонкая разница: для функции σ ноль является точкой насыщения, то есть если пытаться обучить значение этой функции в ноль, вход будет стремиться к минус бесконечности, а производная — к нулю, это стабильное состояние. А для $\tanh$ ноль — это как раз самая нестабильная промежуточная точка, от нуля легко оттолкнуться и начать менять аргумент в любую сторону. О том, почему это важно, мы поговорим в разделе 4.2. Гиперболический тангенс часто используется в некоторых приложениях нейронных сетей, в частности в компьютерном зрении.

В качестве функции активации можно рассмотреть и обычную ступенчатую функцию, она же функция Хевисайда:

$$\mathrm{step}(x) = \begin{cases} 0, & \text{если } x < 0, \\ 1, & \text{если } x > 0. \end{cases}$$

Эта функция использовалась в ранних конструкциях перцептронов. То, что она не определена в нуле, не очень мешает вести обучение: ее можно доопределить, например, как $\mathrm{step}(x) = \frac{1}{2}$, да и на практике случайно попасть точно в ноль вряд ли получится. Один перцептрон со ступенчатой функцией активации обучить вполне возможно. Для этого достаточно просто точно так же подсчитывать

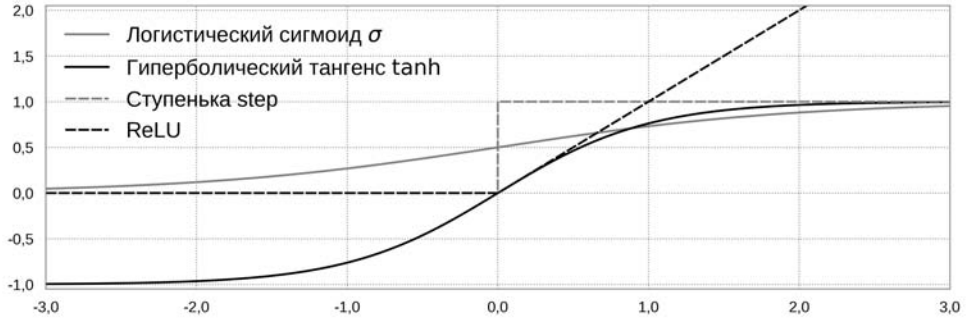


Рис. 3.4. Различные функции активации

«мягкий» результат в виде комбинации входов и весов, но затем превращать его в «жесткое» решение: если «мягкий» результат меньше нуля, выдаем один ответ, если больше нуля, — другой. На последнем шаге классификатора это совершенно нормально, и процессу обучения не мешает.

Но сеть с несколькими уровнями на ступенчатых функциях активации, к сожалению, не построишь, ведь производная от ступеньки просто всегда равна нулю (кроме, опять же, нуля, где она не определена). Таким образом, в нейронной сети из перцептронов со ступенчатыми функциями активации градиенты не дойдут от выходов к входам: по дороге градиент будет умножаться на производную функции step, и ничего кроме нулей не получится...

Разобравшись с классическими функциями активации, давайте двигаться к современности. Здесь тоже есть о чем рассказать. Главная идея, во многом изменившая архитектурные основы современных нейронных сетей, — это так называемые *rectified linear units* (ReLU). Функция активации у них кусочно-линейная:

$$\mathrm{ReLU}(x) = \begin{cases} 0, & \text{если } x < 0, \\ x, & \text{если } x \geq 0. \end{cases}$$

То же самое можно записать более кратко: $\mathrm{ReLU}(x) = \max(0, x)$. И снова мы должны сказать, что это не вполне современная идея: такие искусственные нейроны использовались еще в начале 1980-х годов в модели многоуровневых сетей Кунихико Фукусимы для распознавания образов, получившей название *Neocognitron* [166, 167].

Однако потом от них надолго отказались, и возрождение ReLU произошло уже в разгар революции глубокого обучения; желающим углубиться в детали мы предложим работы [382], где ReLU-активация подробно мотивирована и описана в контексте ограниченных машин Больцмана, и [180], где практическая польза ReLU становится очевидной и в «обычных» нейронных сетях. А сами кратко пройдемся по основным идеям, которые приводят ReLU-нейроны к успеху.

Прежде всего отметим, что ReLU-нейроны эффективнее основанных на логистическом сигмоиде и гиперболическом тангенсе. Например, чтобы подсчитать производную $\sigma'(x)$, нужно вычислить непростую функцию σ, а затем умножить $\sigma(x)$ на $1 - \sigma(x)$; с тангенсом примерно та же история, только нужно возводить в квадрат. А чтобы вычислить производную $\mathrm{ReLU}'(x)$, нужно ровно одно сравнение: если x меньше нуля, выдаем ноль, если больше нуля, — единицу. На первый взгляд это кажется несущественным, но на практике означает, что основанные на ReLU-нейронах сети при одном и том же «вычислительном бюджете» на обучение, на одном и том же «железе» могут быть значительно больше (по размеру, то есть по числу нейронов), чем сети с более сложными функциями активации. Однако сам по себе этот аргумент мало что значит: в конце концов, чтобы подсчитать производную ступенчатой функции, даже сравнивать ни с чем не надо. Нужно еще, чтобы новая конструкция работала и действительно чему-то обучалась.

Чтобы как следует промотивировать ReLU-активацию, объяснить, откуда она берется, нам потребуется одна интересная промежуточная конструкция. Представьте себе активацию перцептрона обычным логистическим сигмоидом. Это хорошая, гладкая функция, и она прекрасно умеет отличать «достаточно активированные» нейроны, когда результат активации становится близким к единице, от «недостаточно активированных», когда этот результат близок к нулю. Но при этом логистический сигмоид дает хоть и непрерывный, но по сути бинарный ответ: например, ему сложно будет отличить активацию «с силой 5» ($\sigma(5) \approx 0{,}9933$) от активации «с силой 10» ($\sigma(10) \approx 0{,}99995$). Фактически это такая сглаженная бинарная ступенька: аргумент функции либо способен на нее «запрыгнуть», либо нет, а оставшийся «запас» сигмоид не интересует. Однако он может быть интересен в том случае, если мы хотели бы различать «сильно активированные» и «немножко активированные» нейроны.

Для этого можно построить любопытную конструкцию — сумму нескольких логистических сигмоидов. Давайте рассмотрим такую функцию активации:

$$f(x) = \sigma\left(x + \frac{1}{2}\right) + \sigma\left(x - \frac{1}{2}\right) + \sigma\left(x - \frac{3}{2}\right) + \sigma\left(x - \frac{5}{2}\right) + \dots.$$

Это сумма бесконечного ряда логистических сигмоидов, каждый из которых смещен вправо относительно предыдущих на единицу. На рис. 3.5 изображены шесть таких сигмоидов, и их сумма (черная кривая) стремится к шести. А что будет в сумме бесконечного ряда? Слева от нуля она ведет себя примерно так же, как один сигмоид: если аргумент x меньше нуля, особенно если значительно меньше, все сигмоиды достаточно близки к нулю, и даже сумма ряда будет маленькой. А вот справа от нуля наблюдается любопытный эффект: сигмоиды, центр которых расположен дальше, правее x, будут по-прежнему близки к нулю (и ряд будет сходиться в любой конечной точке), а сигмоиды, центр которых расположен слева, будут близки к единице. То есть активация этого «стека» сигмоидов примерно подсчитывает вход x.

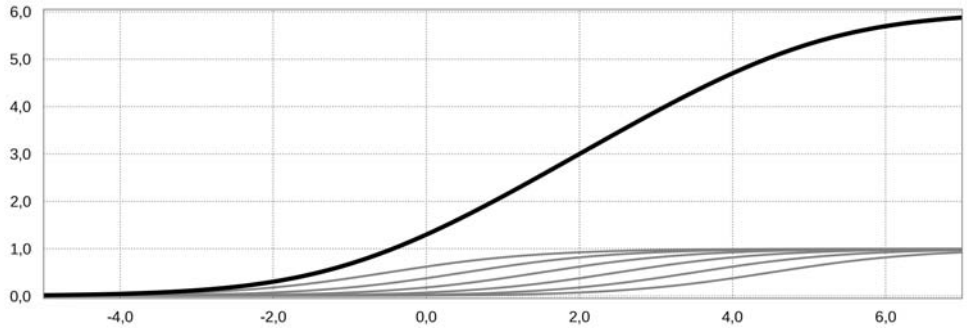


Рис. 3.5. Сумма шести сигмоидов

Если же говорить формально, то нужно сначала заметить, чему равен интеграл сигмоида (проверьте это дифференцированием):

$$\int \sigma(x)\mathrm{d}x = \log\left(1 + e^x\right) + C.$$

И теперь мы видим, что $f(x)$ — это в точности риманова сумма такого интеграла:

$$\int_{1/2}^{\infty} \sigma\left(x + \frac{1}{2} - y\right) \mathrm{d}y,$$

его приближение площадями прямоугольников ширины 1 с центрами в натуральных точках; на рис. 3.5 показано, как это работает. Значит, сумма будет давать приближение к значению интеграла:

$$f(x) = \sum_{i=0}^{\infty} \sigma\left(x + \frac{1}{2} - i\right) \approx \int_{1/2}^{\infty} \sigma\left(x + \frac{1}{2} - y\right) \mathrm{d}y = \\ = \left[-\log\left(1 + e^{x+\frac{1}{2}-y}\right)\right]_{y=1/2}^{y=\infty} = \log\left(1 + e^x\right).$$

Что же мы получили? Оказывается, бесконечный ряд сигмоидов, который гораздо более выразителен, чем один сигмоид, и может выразить понятие «силы активации», — это приблизительно то же самое, что обычная функция $\log\left(1 + e^x\right)$. Ну а эта функция, в свою очередь, очень похожа на $\mathrm{ReLU}(x) = \max(0, x)$ (см. рис. 3.4).

Иначе говоря, если посмотреть на ReLU-нейрон теоретически, мы увидим, что это неплохое приближение к конструкции, которая сильнее и более выразительна, чем обычный логистический сигмоид. Вероятно, именно этим объясняется успех ReLU-активации в современных нейронных сетях.

Есть и менее формальное, но тоже любопытное объяснение: оказывается, такая структура активации гораздо точнее отражает то, что происходит с настоящими нейронами в реальном человеческом (и не только) мозге.

Во-первых, как известно, мозг — это ужасно энергоемкий орган, он потребляет около 20 % всей энергии, которую тратит человек, при собственной массе около 2 % от массы тела. Поэтому для нашего «компьютера» в голове энергоэффективность, «счет за электричество» не менее важны, чем собственно вычислительная мощь: если бы мозг тратил еще больше энергии, человеку, особенно на ранних этапах нашего эволюционного развития, пришлось бы заниматься исключительно тем, что целый день непрерывно искать для мозга еду, и времени собственно подумать уже не осталось бы.

Для имеющегося у него огромного числа нейронов мозг крайне энергоэффективен; это достигается, в частности, *разреженностью* активации нейронов: в каждый момент времени активированы от 1 до 4 % нейронов в мозге [319]. Но если бы они имели сигмоид-активацию и инициализировались случайно, как во многих нейронных сетях, в каждый момент времени заметно активирована бы была где-то половина нейронов; а с правильно регуляризованной ReLU-активацией (о регуляризации мы будем много говорить ниже) несложно достичь высоких показателей разреженности.

Во-вторых, прямые непосредственные исследования функции активации в реальных нейронах и приближенные к биологии модели [105] дают функцию, гораздо больше похожую на ReLU, чем на сигмоид. Это функция интенсивности (частоты срабатывания) выходных сигналов нейрона в зависимости от силы тока на входе; формально говоря, она выглядит так:

$$f(I) = \begin{cases} \left(\tau \log \frac{E+RI-V_{\text{reset}}}{E+RI-V_{th}}\right)^{-1}, & \text{если } E + RI > V_{th}, \\ 0, & \text{если } E + RI \leq V_{th}, \end{cases}$$

где V_{th} — пороговый потенциал активации; V_{reset} — потенциал покоя; I — сила тока на входе; R, E и τ — параметры мембраны нейрона (сопротивление, потенциал и временная константа). График этой функции для реалистичного набора параметров [105] представлен на рис. 3.6. Как видно (и математически это тоже легко доказать), с ростом входа I функция активации быстро становится фактически линейной, и ReLU-активация (с точностью до сдвига) куда больше похожа на то, что происходит в реальной биологической жизни, чем сигмоид.

Итак, мы подробно рассмотрели три основные функции активации: логистический сигмоид, гиперболический тангенс и ReLU. Осталось совсем немного — изучить различные их модификации.

Первая мысль: раз уж мы мотивировали ReLU как приближение к функции $\log(1 + e^x)$, так, может быть, нужно просто взять саму эту функцию, получившую название SoftPlus, как функцию активации, и все станет еще лучше?

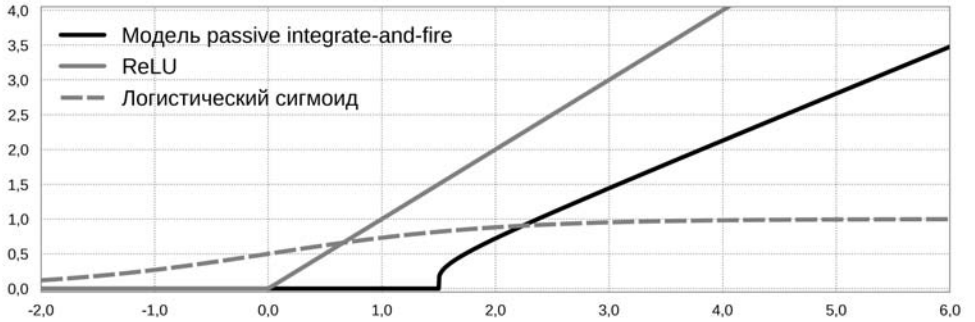


Рис. 3.6. Функция активации живых нейронов, логистический сигмоид и ReLU

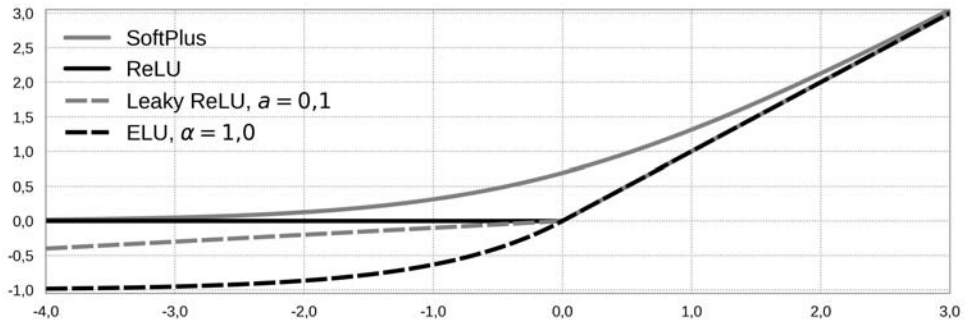


Рис. 3.7. Вариации на тему ReLU: протекающий ReLU, параметризованный ReLU и экспоненциальный линейный нейрон

Ее рассматривали, например, в уже упоминавшейся работе [180], но пришли к выводу, что большого выигрыша нет, а эффективность уменьшается (дифференцировать эту функцию ничем не приятнее, чем тот же логистический сигмоид).

Следующие идеи, которые начали завоевывать популярность в последнее время, — это различные модификации и обобщения ReLU, которые пытаются сохранить вычислительную эффективность, но при этом добавить немного гибкости в базовую конструкцию. Например, *«протекающий ReLU»* (Leaky ReLU, LReLU) [342] обобщает ReLU так: давайте на положительных аргументах оставим функцию той же самой, а на отрицательных отойдем от строгого нуля и сделаем функцию тоже линейной и при этом немножко отрицательной:

$$\mathrm{LReLU}(x) = \begin{cases} ax, & \text{если } x < 0, \\ x, & \text{если } x > 0, \end{cases}$$

где a — это небольшая положительная константа, например $a = 0{,}1$ (см. рис. 3.7).

Таблица 3.1. Различные функции активации: сводная таблица

Название функции	**Формула** $f(x)$	**Производная** $f'(x)$
Логистический сигмоид σ	$\frac{1}{1+e^{-x}}$	$f(x)\left(1-f(x)\right)$
Гиперболический тангенс tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$1-f^2(x)$
SoftSign	$\frac{x}{1+\lvert x\rvert}$	$\frac{1}{(1+\lvert x\rvert)^2}$
Ступенька (функция Хевисайда)	$\begin{cases}0, & x<0\\ 1, & x\geq 0\end{cases}$	0
SoftPlus	$\log\left(1+e^x\right)$	$\frac{1}{1+e^{-x}}$
ReLU	$\begin{cases}0, & x<0\\ x, & x\geq 0\end{cases}$	$\begin{cases}0, & x<0\\ 1, & x\geq 0\end{cases}$
Leaky ReLU, Parameterized ReLU	$\begin{cases}ax, & x<0\\ x, & x\geq 0\end{cases}$	$\begin{cases}a, & x<0\\ 1, & x\geq 0\end{cases}$
ELU	$\begin{cases}\alpha\left(e^x-1\right), & x<0\\ x, & x\geq 0\end{cases}$	$\begin{cases}f(x)+\alpha, & x<0\\ 1, & x\geq 0\end{cases}$

Идея здесь заключается в том, чтобы попытаться улучшить оптимизацию, ведь если на отрицательной части области определения все градиенты строго равны нулю, эти нейроны не будут обучаться вовсе. В [342] показано, что в построенных на глубоких сетях акустических моделях LReLU улучшают распознавание речи.

Дальнейшим развитием этой идеи стал *параметризованный ReLU* (Parametric ReLU, PReLU) [111]. Эта конструкция выглядит точно так же, как и LReLU, с той лишь разницей, что теперь константу a тоже можно обучать для каждого конкретного датасета по-своему; в результате получается строго более выразительная система, чем обычный ReLU и LReLU, и если веса, включая константу, правильно инициализировать и обучать, то результаты станут лучше [114].

Другой вариант модификации — это *экспоненциальный линейный нейрон* (Exponential Linear Unit, ELU) [86], в котором на отрицательных значениях аргумента функция активации становится экспоненциальной:

$$\mathrm{ELU}(x)=\begin{cases}\alpha\left(e^x-1\right), & x<0,\\ x, & x\geq 0.\end{cases}$$

Идея здесь состоит в том, чтобы сочетать в одной функции наличие отрицательных значений (что важно для обучения) и их быстрое насыщение при дальнейшем уменьшении аргумента (это важно, чтобы сохранить разреженность). Есть и другие варианты функций активации, которые тоже в некоторых областях и экспериментах показывают себя лучше существующих; в целом, это очень активно развивающаяся область исследований в современном глубоком обучении.

Все те функции, о которых мы говорили в этом разделе, а также несколько других (тоже иногда используемых на практике), показаны в табл. 3.1. Что же выбрать бедному разработчику, как оценить, что из этого многообразия лучше подходит для конкретной задачи?

На самом деле это далеко не первый и не главный вопрос в разработке реальной системы глубокого обучения; архитектура системы и алгоритмы оптимизации обычно гораздо важнее. Подавляющее большинство примеров в этой книге будет использовать одну из двух функций активации: либо логистический сигмоид σ, либо ReLU. С них, особенно ReLU, мы и рекомендуем начинать разработку, а потом уже, если останется время, можно попробовать параметризованные ReLU и другие современные идеи: они могут дать некоторое улучшение качества, но оно, вероятнее всего, будет маргинальным, и эту оптимизацию лучше отложить на потом.

3.4. Как же обучаются настоящие нейроны

> ...Логическое мышление даже у «цивилизованных» людей похоже, скорее, на танцы лошадей, то есть на трюк, которому можно обучить некоторых, но далеко не всех, причём он может исполняться лишь с большой затратой сил и с разной степенью мастерства, и даже лучшие представители не в состоянии повторять его много раз подряд.
>
> *Х. Й. Ульдалль. Основы глоссематики*

В этой книге мы рассмотрим массу различных архитектур для нейронных сетей: сверточные сети, рекуррентные, сети с вниманием, сети с памятью, такие сети, сякие, этакие... Когда мы будем описывать новые архитектуры, мы еще не раз вернемся к устройству человеческого мозга и вынесем из его естественной архитектуры какие-то новые идеи о том, как нам устроить архитектуру сетей искусственных.

Но при этом мы всю дорогу будем обучать сети с помощью методов, основанных на градиентном спуске, суть которого мы уже рассмотрели в разделе 2.4, а о современных модификациях поговорим в разделе 4. Алгоритмически градиентный спуск реализуется через обратное распространение ошибки: мы постепенно считаем градиент сложной композиции элементарных функций и передаем эти градиенты по сети в обратном направлении.

Действительно ли настоящие, «живые» нейроны работают именно так? Скорее всего, нет. Есть сразу несколько серьезных причин считать, что обратное распространение ошибки биологические нейроны реализовать не могут.

Во-первых, они передают друг другу не вещественные числа, а бинарные сигналы: один спайк, одна активация нейрона не содержит достаточно информации для того, чтобы передать значения градиентов (на самом деле, конечно, все немного сложнее, но все-таки). Но это само по себе не беда: говоря о дропауте в разделе 4.1,

мы обсудим, зачем биологическим нейронам может понадобиться передавать не сам градиент, а его стохастический и сильно искаженный вариант. Есть, например, яркий результат, говорящий о том, что достаточно (стохастически) передавать один бит информации на прямом проходе вычисления значений сети и два бита на обратном, чтобы обратное распространение ошибки работало достаточно хорошо.

Во-вторых, у нейрона нет соединений, работающих в обе стороны сразу: нейрон передает сигналы по аксону и получает входы через дендриты, а автоматических двунаправленных связей, которые можно было бы использовать в одну сторону при прямом проходе и в другую при обратном, у него нет.

В-третьих, и это самое серьезное возражение, для обратного распространения ошибки необходимо, чтобы нейрон умел передавать *два разных* типа значений: во время прямого прохода он должен передать результат вычисления своей функции от входов, а во время обратного — результат вычисления градиента своей функции по входам.

Для компьютера это, конечно, совершенно не проблема: где один алгоритм, там и два. А вот для биологических нейронов это становится непреодолимым препятствием: в них нет механизма, который позволял бы работать по-разному в зависимости от чего бы то ни было, кроме собственно своих входов. Поэтому, хотя до конца это еще не ясно, биологические нейроны вряд ли могут непосредственно реализовывать обратное распространение ошибки[1]. Но какие алгоритмы они реализуют на самом деле? Может быть, мы можем позаимствовать у природы не только архитектуры, но и алгоритмы?

В начале этой главы мы уже упоминали основную альтернативную идею обучения нейронов в живом мозге: *обучение по Хеббу*. Эта идея появилась в конце 1940-х годов [209] и основана на простом принципе: те связи, которые часто активируются, получают большие веса, а те, которые активируются редко, постепенно отмирают. Идеи Хебба быстро стали мейнстримом нейробиологии, послужили одним из оснований бихевиоризма, и до сих пор считаются верными, периодически только уточняясь. Математически суть такого обучения проста: предположим, что у нас есть набор нейронов, которые как-то связаны друг с другом, и каждая связь между нейронами i и j характеризуется весом w_{ij}. Подадим на входы сети некий тренировочный пример; в результате нахождения значений активаций каждый нейрон будет вычислять некоторый выход x_i. Тогда обучение будет состоять в том, чтобы увеличить веса связей между активированными парами нейронов:

$$\Delta w_i = \eta x_i x_j.$$

В этом уравнении можно либо считать, что каждый x_i равен или нулю, или единице, и увеличивать веса только активированных нейронов, либо считать, что x_i равны 1 и -1, и при этом увеличивать также веса между нейронами, ни один из которых не активирован. Кроме того, нужно либо понемножку уменьшать веса

[1] На этот счет, впрочем, есть и другие идеи: см., например, доклад Джеффри Хинтона «Как сделать обратное распространение в мозге» [218]. Однако показательно то, что этот доклад так и не превратился даже в статью.

всех нейронов равномерно, либо перенормировать веса так, чтобы общая «энергия» системы оставалась постоянной.

Правило обучения Хебба выглядит гораздо проще и логичнее для биологических нейронов, чем обратное распространение градиентов. Однако в 1940-х годах оно тоже сильно опередило развитие нейробиологии. Первым примером биологического механизма, который поддерживает обучение по Хеббу, стала долговременная потенциация (long-term potentiation, LTP). При ней как раз усиливается синаптическая передача между двумя нейронами, которые активируются одновременно. Открыли ее Терье Лемо и Тимоти Блисс, проводившие эксперименты на кроликах в конце 1960-х — начале 1970-х годов [49, 50, 332], и сейчас считается, что долговременная потенциация — это один из основных механизмов того, как именно работает память и обучение.

В искусственном интеллекте правило Хебба привело к появлению так называемых *сетей Хопфилда*. Это графические вероятностные модели, которые реализуют ассоциативную память: сеть (по форме она обычно представляет собой двумерную решетку) «запоминает» несколько желаемых состояний-ассоциаций. Запоминание происходит с помощью обучения по Хеббу, и в результате такого обучения запомненные состояния становятся локальными минимумами энергии сети. Сеть затем должна сходиться к одному из них из любого начального состояния. Идея сетей Хопфилда состояла в том, чтобы таким образом моделировать процессы ассоциативного вспоминания, когда возможных ассоциаций несколько, и всплывает «самая близкая» из них [230, 451].

Однако пока сети Хопфилда работают совсем не так хорошо, как хотелось бы. Есть и современные нейронные сети с памятью (мы кратко обсудим их в главе 8), но представляется, что основные исследования в области моделирования ассоциативной памяти еще впереди.

В современной нейробиологии явление долговременной потенциации уже, конечно, изучено в подробностях. В частности, нейробиологи выяснили, что усиление синаптической связи между нейронами зависит не только от самого факта совместной активации, но и от конкретного времени, когда все это происходит, то есть от того, какое время проходит между спайками нейронов на входе и на выходе. Это явление получило название spike-timing-dependent plasticity (STDP) [70, 352]. Недавняя работа Йошуа Бенджи с соавторами [538] пытается дать теоретическое обоснование STDP в контексте глубокого обучения, с точки зрения машинного обучения и оптимизации функций.

Есть и другие идеи о том, как можно обучать нейронные сети без всяких градиентов или с градиентами, выраженными в очень неожиданной форме. Так, Джеффри Хинтон и Джеймс Макклелланд еще во второй половине 1980-х годов предложили способ обучать автокодировщики без распространения ошибки, за счет так называемой рециркуляции активаций по нейронной сети [222]. Давайте рассмотрим ее на простейшем примере, когда в сети есть группа видимых нейронов, на которые подаются входы, и группа скрытых нейронов, на которых должно обучиться хорошее представление этих входов. Граф связей двудольный: каждый

видимый нейрон связан с каждым скрытым, но не между собой. Такая архитектура очень похожа на ограниченную машину Больцмана, и алгоритм ее обучения тоже будет чем-то похож. Пусть, более того, каждый нейрон — это обычный знакомый нам перцептрон с сигмоидальной функцией активации:

$$y_j = \sigma\left(z_j\right) = \sigma\left(\sum_i x_i w_{ji} - b_j\right).$$

Тогда процедура рециркуляции работает следующим образом. Представим себе, что обучение нейронной сети происходит не абстрактно, а «в реальном времени», последовательной передачей активаций между нейронами, и передача активаций идет в обе стороны: в каждый (дискретный) момент времени активации нейронов меняются.

Тогда то, что вернется к нейрону видимого уровня после путешествия туда-обратно по сети, — это и есть восстановленный вход, который у автокодировщика должен быть как можно ближе к изначальному входу. И та самая производная, в сторону которой нужно изменять веса на пути от нейронов скрытого уровня к видимым, — это всего лишь разница между тем, что видимый нейрон посылал раньше, и тем, что пришло к нему обратно! Осталось только предположить, что каждый нейрон способен «запоминать» свое предыдущее значение и действовать на основе разницы между своим предшествующим состоянием и вновь пришедшей активацией. Это предположение биологически вполне разумно. Получается, что изменение веса от j-го нейрона скрытого уровня к i-му нейрону видимого уровня можно подсчитать как

$$\Delta w_{ij} = \eta y_j^{(1)}\left(y_i^{(0)} - y_i^{(2)}\right),$$

где верхние индексы обозначают время. В [222] оказалось, что такое правило обновления действительно сходится куда надо, просто сходится несколько хуже, чем обычный градиентный спуск, поэтому для искусственных нейронных сетей нет большого смысла применять такое обучение.

Несмотря на эти отдельные работы, в данном разделе, как и ранее в разделе 1.4, мы увидели, что человеческий мозг и нейронные сети — это хотя и чем-то похожие, но все-таки пока очень разные устройства обработки информации. Поэтому современные нейронные сети не стоит рассматривать как попытку эмулировать работу мозга: да, мы иногда вдохновляемся тем, как в наших головах все устроено, но пока и не очень-то понимаем, как мозг обучается на самом деле, и не можем точно эмулировать даже то, что уже понимаем.

Еще раз мы вернемся к этой теме в разделе 5.1, где поговорим о том, как устройство участков коры головного мозга, отвечающих за зрение, помогло исследователям придумать сверточные нейронные сети. Но в целом тему соответствия между нейронными сетями и человеческим мозгом пора закрыть: отныне для нас нейронные сети — это формализм машинного обучения, а не попытка что-то подсмотреть у природы.

3.5. Глубокие сети: в чем прелесть и в чем сложность?

> Если у меня было какое-нибудь желание, то лишь одно: глубже, сильнее погрузиться в этот сон, все глубже и глубже, и еще глубже...
>
> *А. Гарборг. Смерть*

> We need to go deeper.
>
> *Из к/ф Inception*

К настоящему моменту мы узнали уже очень много о нейронных сетях. Мы поняли, что они состоят из нейронов, разобрали, как выглядит один нейрон, как они соединяются в единую нейронную сеть и, главное, как потом можно обучать эту нейронную сеть: для этого используются разные варианты градиентного спуска, которые мы подробно рассмотрим в главе 4, а чтобы собственно подсчитать градиент, можно воспользоваться алгоритмами автоматического дифференцирования на графе вычислений, о которых мы недавно говорили. Все эти методы, вообще говоря, никак не зависят от архитектуры нейронной сети и теоретически должны работать для любого графа вычислений, лишь бы мы умели пробрасывать градиенты через узлы, то есть реализовывать процедуру обратного распространения.

Идея графа вычислений не кажется такой уж сложной, о том, как дифференцировать композицию функций, люди знают уже на протяжении нескольких веков, да и непосредственно алгоритм градиентного спуска был известен даже раньше XX века. Сама идея построить глубокую сеть, то есть нанизать друг на друга несколько уровней нейронов, тоже не кажется верхом изобретательности. Неужели было так сложно скомбинировать эти идеи вместе, что революция глубокого обучения началась только в XXI веке, в 2005–2006 годах, а до этого глубокие архитектуры и алгоритмы были неизвестны?

Конечно, нет. Идеи глубоких нейронных сетей имеют почти такую же долгую историю, как и сами искусственные нейронные сети[1]. Первые глубокие сети появились еще в середине 1960-х годов, и здесь есть повод для местечковой гордости: первые настоящие глубокие сети в виде глубоких перцептронов были описаны в работах советского ученого А. Г. Ивахненко[2] [257, 588]. Ивахненко разработал

[1] За ранней историей вопроса здесь мы опять обращаемся к ссылкам из уже упоминавшегося исторического обзора Юргена Шмидхубера [475]; кстати, на недавней конференции NIPS 2016 Шмидхубер в своем докладе посвятил отдельный слайд Алексею Григорьевичу Ивахненко, назвав его «отцом глубокого обучения».

[2] *Алексей Григорьевич Ивахненко* (1913–2007) — советский ученый, специалист в области систем управления. Метод группового учета аргументов стал его основным результатом, и подобные методы Ивахненко развивал и применял затем всю жизнь. Хотя его работы получили широкое признание во всем мире [150], сейчас его редко вспоминают среди первооткрывателей глубокого обучения; будем надеяться, что немец Шмидхубер сможет отстоять «славу советского оружия».

так называемый *метод группового учета аргументов* [255], суть которого выглядит примерно так:

- сначала мы выбираем общий вид, параметрическое семейство моделей, которые будем обучать; Ивахненко предлагал использовать так называемые *полиномы Колмогорова — Габора*, то есть по сути просто многочлены с неизвестными коэффициентами, но могут быть и любые другие;
- строим и обучаем разные варианты выбранных моделей;
- выбираем с помощью метрики качества несколько лучших моделей; если нужное качество уже достигнуто, можно ничего дальше не делать;
- но если еще не достигнуто — и это ключевой момент — то мы начинаем строить модели следующего уровня, используя выходы подобранных на предыдущем шаге моделей как входы для последующих;
- этот процесс можно рекурсивно повторять до тех пор, пока качество модели либо не достигнет нужного уровня, либо не перестанет улучшаться.

Метод группового учета аргументов выглядит на удивление современно. Если в нем в качестве базовой модели выбрать перцептрон, мы получим типичную нейронную сеть с несколькими слоями, которая обучается слой за слоем: сначала первый, потом он фиксируется и начинается обучение второго, и т. д. Уже в начале 1970-х годов этим методом вполне успешно обучались модели вплоть до семи уровней в глубину [256], и это очень похоже на процедуру предобучения без учителя, которую мы уже упоминали и подробно обсудим чуть ниже.

Но все-таки нейронные сети в итоге пошли немножко другим путем. Первой глубокой нейронной сетью можно считать уже упоминавшийся *Neocognitron* Кунихиро Фукусимы [166, 167], в котором появились и сверточные сети, и активации, очень похожие на ReLU. Однако эта модель не обучалась в современном смысле этого слова: веса сети устанавливались из локальных правил обучения без учителя. Примерно в то же время появились и глубокие модели на основе обратного распространения. Первым применением обратного распространения ошибки к произвольным архитектурам можно считать работы финского тогда еще студента Сеппо Линненмаа [329][1]: в 1970 году он построил правила автоматического дифференцирования по графу вычислений, которые мы рассматривали в разделе 2.5, в том числе и обратное распространение. Однако нейронными сетями и вообще машинным обучением Линненмаа тогда не интересовался. К ним эти идеи были применены в работах Дрейфуса [128] и Вербоса [564], а начиная с классических работ Румельхарта, Хинтона и Уильямса, увидевших свет в 1986 году и чуть позже [459, 460], метод обратного распространения стал общепринятым для обучения любых нейронных архитектур. На этом история нейронных сетей начинает

[1] Целую биографическую сноску писать о Линненмаа не будем, но отметим, что он получил самую первую степень Ph.D. по информатике в истории университета Хельсинки. А работу об обратном распространении он защитил в качестве диплома магистра (M.Sc.)! Читающим нас студентам посоветуем не вешать нос и не считать, что «тогда было интереснее»: поверьте, на ваш век интересных задач тоже хватит.

заметно разветвляться, и мы вернемся к ней уже в соответствующих главах, посвященных конкретным архитектурам.

А сейчас перейдем к вопросу, так сказать, телеологического характера: зачем вообще нужны глубокие сети? Классическая теорема Хорника [231], основанная на более ранних работах Колмогорова [589], утверждает, что любую непрерывную функцию можно сколь угодно точно приблизить нейронной сетью с одним скрытым уровнем. Казалось бы, этого должно быть достаточно. Зачем плодить лишнюю сложность на ровном месте?

Дело в том, что глубокие архитектуры часто позволяют выразить то же самое, приблизить те же функции гораздо более эффективно, чем неглубокие. Читатели с некоторой подготовкой в теоретической информатике могут, прежде чем двигаться дальше, вспомнить аналогичные утверждения о булевых схемах: известно, что схемы глубиной $k + 1$ могут эффективно выразить строго больше булевских функций, чем схемы глубиной k. Более того, можно даже построить экспоненциальные разделения между разным числом уровней, то есть для всякого k можно придумать функцию, которую можно выразить полиномиальной схемой глубины $k + 1$, но которая требует экспоненциального размера схемы глубины k [208]. С уровнями нейронной сети возникает тот же эффект: одну и ту же функцию часто можно гораздо лучше приблизить более глубокой сетью, чем мелкой, даже если общее число нейронов в сети оставить постоянным.

Мы не будем вдаваться здесь в геометрические подробности, но рекомендуем на эту тему работу [398], в которой очень изящно показано, что слой нейронов с ReLU-активацией фактически «сворачивает» пространство, отождествляя некоторые его части между собой; и поэтому разделяющие поверхности, построенные в этом «свернутом» пространстве, потом «разворачиваются» в гораздо более сложные конструкции в пространстве собственно входных векторов.

А другая сторона силы глубоких сетей — это то, что глубокая нейронная сеть создает не просто глубокое, но еще и *распределенное представление*. Здесь речь идет о том, что каждый уровень глубокой сети состоит не из одного нейрона, а сразу из многих, и комбинации значений этих нейронов производят самый настоящий экспоненциальный взрыв в пространстве входов!

Мы проиллюстрировали это примером на рис. 3.8, подобный которому часто приводит в своих статьях и книгах Йошуа Бенджи [38, 184]. Представим себе, что мы можем разделять точки на плоскости с помощью трех возможных признаков, каждый из которых — это линейная функция: f_1, f_2, f_3. На рис. 3.8, *а* изображена разделяющая поверхность по одному из них, то есть прямая на плоскости, и размечены части, на которые прямая делит плоскость. Частей этих, разумеется, две.

Как следует объединять эти признаки в нашем классификаторе? Во-первых, можно попробовать добавить глубины и сделать дерево принятия решений; пример такого дерева и соответствующая разделяющая поверхность показаны на рис. 3.8, *б*. Обратите внимание, что дерево делит плоскость на столько же частей,

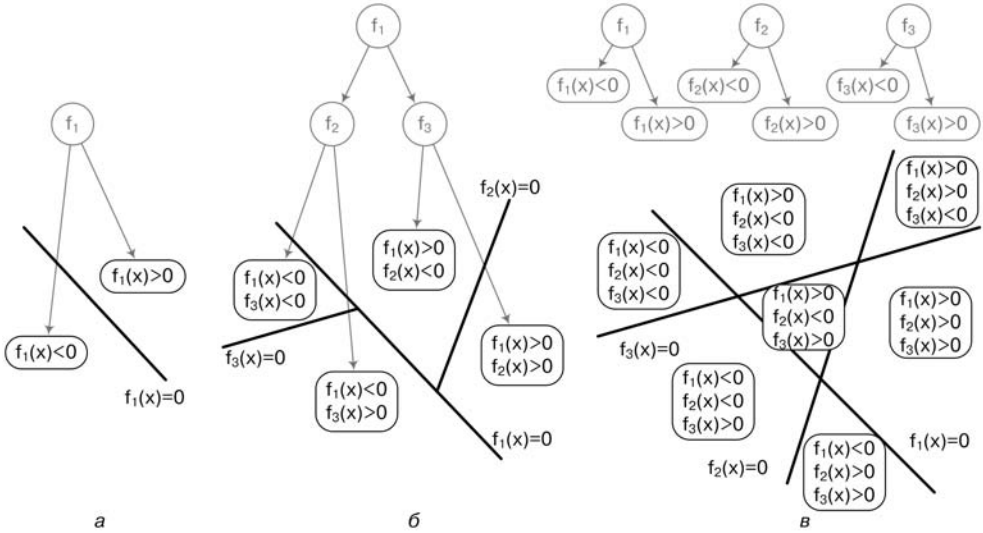


Рис. 3.8. Сила распределенных представлений: *а* — разделяющая поверхность одного признака; *б* — разделяющая поверхность дерева глубины 2 на трех признаках; *в* — разделяющая поверхность трех признаков сразу

сколько у него листьев: чтобы разобрать четыре разных возможных случая, нам потребовалось построить дерево с четырьмя листьями. А на рис. 3.8, *в* изображено распределенное представление: давайте представим, что наш «второй слой» может состоять из трех нейронов сразу. Тогда всевозможные комбинации этих трех нейронов могут распознать сразу $2^3 = 8$ разных случаев (на картинке из них реализуется 7), и это число разных вариантов растет экспоненциально при росте числа нейронов.

А теперь представьте (нарисовать это было бы уже сложно — слишком запутанная поверхность получилась бы), что за вторым уровнем есть еще третий, то есть мы можем составлять разные комбинации из таких вот разделяющих поверхностей по три прямые в каждой...

Ну хорошо. Предположим, мы вас убедили в том, что глубокие нейронные сети действительно нужны. Но тогда возникает второй вопрос: а в чем, собственно, проблема? Почему глубокие сети, в которых нет ничего теоретически сложного и которые, как мы уже видели, появились почти одновременно с алгоритмом обратного распространения ошибки, так долго вызывали такие сложности? Почему революция глубокого обучения произошла в середине 2000-х, а не 1980-х годов?

На этот вопрос есть два ответа. Первый — математический. Дело в том, что глубокие нейронные сети обучать, конечно, можно тем же алгоритмом градиентного спуска, но в базовом варианте, без дополнительных ухищрений работать это не

будет. Представьте себе, что вы начали обучать глубокую сеть алгоритмом обратного распространения ошибки. Последний, ближайший к выходам уровень обучится довольно быстро и очень хорошо. Но что произойдет дальше? Дальше окажется, что большинство нейронов последнего уровня на всех тестовых примерах уже «определились» со своим значением, то есть их выход близок или к нулю, или к единице. Если у них классическая сигмоидальная функция активации, то это значит, что производная у этой функции активации близка к нулю с обеих сторон... но ведь на эту производную мы должны умножить все градиенты в алгоритме обратного распространения!

Таким образом, получается, что обучившийся последний слой нейронов «блокирует» распространение градиентов дальше назад по графу вычислений, и более ранние уровни глубокой сети в результате обучаются очень медленно, фактически стоят на месте. Этот эффект называется *проблемой затухающих градиентов* (vanishing gradients).

А в главе 6 мы увидим, что с рекуррентными сетями, которые фактически по определению являются очень глубокими, возникает еще и обратная проблема: иногда градиенты могут начать «взрываться», экспоненциально увеличиваться по мере «разворачивания» нейронной сети (exploding gradients). Обе проблемы возникают часто, и достаточно долго исследователи не могли их удовлетворительно решить.

В середине 2000-х годов появились первые действительно хорошо работающие и хорошо масштабирующиеся конструкции... и они очень сильно напоминали исходный механизм «глубоких моделей» Ивахненко! Решение, которое предложили в группе Хинтона [223, 465, 466], заключалось в том, чтобы предобучать нейронные сети уровень за уровнем с помощью специального случая ненаправленной графической модели, так называемой *ограниченной машины Больцмана* (restricted Boltzmann machine, RBM). Дело в том, что градиентный спуск, конечно, хорошо находит локальный минимум функции, а разные умные варианты градиентного спуска (см. разделы 4.4 и 4.5) способны даже выбираться из небольших локальных минимумов и приходить в более значительный минимум. Но все равно градиентный спуск по сути своей *локален*, он делает лишь небольшие шаги в ту или иную сторону от текущей точки, и результат сильно зависит от инициализации, от того, с какой точки мы будем начинать. Поэтому предобучение без учителя может привести к очень хорошему эффекту: за счет того, что модель уже начинает потихоньку «разбираться» в структуре предлагаемых данных, начальные значения весов уже не случайные, а достаточно разумные. Такие конструкции привели к прорыву сначала в распознавании речи, а затем и в обработке изображений...

Хмм... подождите-ка. Что-то здесь не сходится. С одной стороны, мы пришли к тому, что просто так глубокие сети обучать не удается, и нужно применять разнообразные трюки, сначала последовательно обучая отдельные слои сети совершенно другими алгоритмами, а потом только локально «докручивая» их градиентным спуском. Однако почему-то половина этой книги вовсе не посвящена обучению

машин Больцмана! И это не потому, что они неявно участвуют в обучении каждой глубокой сети — нет, использующиеся для глубокого обучения библиотеки достаточно прозрачны, они просто строят граф вычислений так, как мы их попросим это сделать, а затем буквально считают градиенты. Что произошло, куда пропали все былые трудности?

Во-первых, по сравнению с серединой 2000-х годов, и тем более по сравнению с началом 1990-х, мы сейчас все-таки лучше умеем обучать нейронные сети. Появился ряд важных инструментов, которые можно так или иначе отнести либо к *регуляризации*, либо к разным модификациям *оптимизации* в нейронных сетях. Этим инструментам будет посвящена глава 4: мы поговорим и о собственно регуляризации в том смысле, в котором она появилась в разделе 2.2, и о дропауте, и о нормализации мини-батчей, и о правильной случайной инициализации весов, и о новых модификациях градиентного спуска... Разговор обо всех этих новшествах нам еще предстоит. В целом, хотя основные конструкции нейронных сетей во многом пришли к нам еще из 90-х, а то и 80-х годов прошлого века, обучаем мы их сейчас все-таки не самым наивным способом.

Но и это еще не вся правда. На вопрос о том, почему глубокие нейронные сети было сложно обучать, есть и другой ответ. Он очень простой и может вас разочаровать: дело в том, что раньше компьютеры были медленнее, а доступных для обучения данных было гораздо меньше, чем сейчас.

Для примера из следующего раздела это не так важно, аналогичную сеть можно было обучить и в начале 1990-х годов. Но вот, например, обучение системы распознавания речи выглядело примерно так: исследователи писали код и запускали обучение, которое продолжалось на тогдашних компьютерах недели две. Причем происходило это двухнедельное обучение на классическом датасете TIMIT, который сейчас считается довольно маленьким и используется для быстрых экспериментов. Через две недели смотрели на результат, понимали, что, по всей видимости, нужно было подкрутить тот или иной параметр (а их в нейронных сетях очень много, как мы еще не раз увидим), подкручивали... и опять запускали обучение на две недели. Конечно, такая обстановка не способствовала ни тонкой настройке сетей, ни своевременным публикациям к соответствующим дедлайнам, ни попросту технической возможности успешно обучить нейронную сеть.

В середине 2000-х годов все сошлось воедино: технически компьютеры стали достаточно мощными, чтобы обучать большие нейронные сети (более того, вычисления в нейронных сетях вскоре научились делегировать видеокартам, что ускорило процесс обучения еще на целый порядок), наборы данных стали достаточно большими, чтобы обучение больших сетей имело смысл, а в математике нейронных сетей произошло очередное продвижение. И началась та самая революция глубокого обучения, которой посвящена вся эта книга. В следующем разделе мы приведем конкретный пример нейронной сети, которая в начале 1990-х годов считалась бы передним краем науки и обучалась бы наверняка часами, а сейчас составляет базовый пример применения библиотеки TensorFlow и может обучиться за несколько десятков секунд.

3.6. Пример: распознавание рукописных цифр на TensorFlow

> *Директор.* Сейчас еще что-нибудь сделаем! С психологией кончено, обратимся к графологии... Мне отвечают; я вижу, я уверен, что этот стоит пяти. Я ставлю пять, инстинктивно ставлю уверенно, большую пятерку, во всю клетку! А то, знаете, неуверенность этакая является: «Как будто это стоит пяти?». И я инстинктивно ставлю маленькую этакую пятерку, боязливую. Это, знаете, как гимназист младших классов: не уверен, надо ли поставить запятую, так он ставит маленькую запятую. Считайте только большие пятерки. Маленькие не считаются!
>
> *В. Дорошевич. Конкурс*

Выше мы уже говорили о том, что нейронную сеть можно представить в виде абстрактного направленного графа, вершинами которого будут математические операции и точки входа и выхода для данных и переменных. Эту идею успешно воплотили в код сразу несколько групп исследователей. Первой успешной библиотекой для автоматического дифференцирования, которая на долгие годы определила ландшафт глубокого обучения в мире и во многом продолжает его определять и сейчас, стала библиотека Theano[1] [528, 529] (см. документацию Theano, которая содержит много примеров нейронных сетей и, в том числе, глубоких моделей [108]).

Однако в 2015 году Google анонсировала выход своей собственной библиотеки TensorFlow с открытым исходным кодом [523]. Поскольку на данный момент представляется, что реализацией TensorFlow удобнее пользоваться (в частности, сложные модели в Theano нужно довольно долго компилировать, а в TensorFlow стадии компиляции фактически нет), а также можно ожидать, что исследователи из Google будут продолжать развивать проект и дальше, в этой книге мы будем приводить примеры в основном на TensorFlow. В частности, в настоящем разделе мы рассмотрим автоматическое дифференцирование и обучение нейронных моделей на примере классической задачи распознавания рукописных цифр.

Многие из моделей, встречающихся в этой книге, мы будем обучать на наборе данных MNIST [188, 315]. Это один из самых известных, самых избитых наборов данных как распознавания изображений в целом, так и глубокого обучения в частности. До сих пор многие модели прежде всего обучают именно на MNIST, хотя

---

[1] *Феано*, или *Теано Кротонская* — одна из первых женщин-философов, принадлежавшая к пифагорейской школе. Некоторые источники называют ее женой Пифагора, другие — дочерью. О ней сохранились лишь весьма разрозненные сведения; согласно основным современным версиям, либо в этот образ слились две жившие в разное время женщины, либо имя Феано и вовсе было псевдонимом более поздних авторов, которые хотели применить пифагорейское учение к жизни женщины.

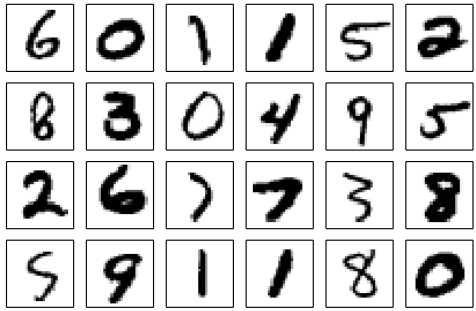


Рис. 3.9. Набор данных MNIST: примеры рукописных цифр.

для современных методов он уже никаких сложностей не представляет[1]. Не обойдем его вниманием и мы. MNIST состоит из рукописных цифр, изображенных на американский манер (см. несколько примеров цифр из MNIST на рис. 3.9). Всего в MNIST содержится 70 000 размеченных черно-белых изображений размером 28×28 пикселов; «размеченных» здесь означает, что каждому изображению в данных уже поставлен в соответствие правильный ответ — цифра, которую хотел изобразить человек на этой картинке.

Поскольку MNIST — это своего рода Hello World для современной обработки изображений, в TensorFlow этот набор данных поддерживается «из коробки», и для импорта MNIST достаточно написать буквально две строчки кода:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Полученные данные уже разбиты на тренировочную (`mnist.train`), тестовую (`mnist.test`) и валидационную (`mnist.validate`) выборки, содержащие 55 000, 10 000 и 5000 примеров соответственно. Каждая из этих выборок состоит из изображений цифр (`mnist.train.images`) и их меток (`mnist.train.labels`); рис. 3.9 показывает несколько образцов изображений с различными метками.

В этом примере мы не будем использовать двумерную структуру изображений, а представим их в виде обычных векторов размерности 784. В главе 5 мы еще вернемся к этому примеру и покажем, как правильно пользоваться двумерным расположением пикселов на картинке и насколько это получится эффективнее, однако сейчас для наших нужд достаточно и простой одномерной задачи.

Таким образом, тренировочная выборка `mnist.train.images` может быть представлена в виде матрицы размерности $55\,000 \times 784$. Отметим, что и в TensorFlow, и в theano часто требуются не только матрицы, но и многомерные представления

[1] Джеффри Хинтон назвал MNIST «дрозофилой машинного обучения», имея в виду, что MNIST дает изучающим машинное обучение такой же хорошо определенный, небольшой и несложный, но при этом нетривиальный пример, как дрозофила генетикам.

данных — например, тот же MNIST логичнее представить в виде трехмерной «матрицы» размерности $55\,000 \times 28 \times 28$. Такие «многомерные матрицы» в TensorFlow называются *тензорами*; выше мы уже сетовали на то, что тензоры не настоящие, а просто многомерные массивы, но что поделать.

В качестве модели для обучения мы рассмотрим *softmax-регрессию*. Это обобщение логистической регрессии на случай нескольких классов: чтобы получить «вероятности» классов, которые нам хочется оценить, мы применяем так называемую softmax-функцию к вектору получившихся ненормализованных оценок:

$$\text{softmax}(x)_j = \frac{\exp(x_j)}{\sum_i \exp(x_i)}.$$

Идея softmax-функции состоит в том, чтобы несколько заострить, преувеличить разницу между полученными значениями: softmax будет выдавать значения, очень близкие к нулю, для всех x_j, существенно меньших максимального.

В качестве функции потерь мы будем использовать стандартную для логистической регрессии перекрестную энтропию (кросс-энтропию, cross-entropy):

$$H_t(y) = -\sum_i t_i \log y_i,$$

где y — предсказанное нами значение, а t — исходная разметка (правильный ответ). Мы подробно обсуждали перекрестную энтропию в разделе 2.3.

Для удобства метки изображений можно представить в виде так называемых *one-hot vectors* — векторов, в которых единица стоит в позиции, соответствующей исходной метке, а в остальных позициях стоят нули. Например, вектор $[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$ будет соответствовать ответу 3. Тогда `mnist.train.labels` — это тензор (матрица) размера $55\,000 \times 10$.

Обычно в Python для сложных вычислений используются вспомогательные библиотеки численных вычислений, например *NumPy*; такие библиотеки выполняют дорогие операции (например, умножение матриц) не силами самого интерпретатора языка Python, а с помощью оптимизированного кода, написанного на других языках (обычно C или Fortran). К сожалению, даже переключение между операциями в Python и вне его может оказаться слишком дорогим, особенно когда речь идет о вычислениях на GPU. Поэтому вместо того, чтобы выполнять каждую отдельную операцию независимо от Python, TensorFlow предлагает возможность описать в Python граф вычислений, а затем запускать всю модель вне Python.

Чтобы использовать TensorFlow, нужно его сначала импортировать:

```
import tensorflow as tf
```

В TensorFlow требуемые операции выражаются с помощью символьных переменных, поэтому давайте создадим переменную для тренировочных данных:

```
x = tf.placeholder(tf.float32, [None, 784])
```

В данном случае x — это не какой-то заранее заданный тензор, а так называемая *заглушка (placeholder)*, которую мы заполним, когда попросим TensorFlow произвести вычисления. Мы хотим иметь возможность использовать произвольное число 784-мерных векторов для обучения, поэтому в качестве одной из размерностей указываем `None`. Для TensorFlow это значит, что данная размерность может иметь произвольную длину.

Кроме заглушки для тренировочных данных, нам также потребуются переменные, которые мы собственно и будем изменять при обучении нашей модели.

```
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

Здесь W имеет размерность 784×10, так как мы собираемся умножать вектор размерности 784 на W и получать предсказание для 10 возможных меток, а вектор b размерности 10 — это свободный член, bias, который мы добавляем к выходу.

После того как мы импортировали нужные модули и объявили все переменные, собственно нашу модель на TensorFlow можно записать в одну строчку!

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

Сначала мы перемножаем матрицы x и W с помощью `tf.matmul(x,W)`, затем добавляем к результату b, и для получения вероятностей классов применяем `tf.nn.softmax`.

Для того чтобы обучить модель, нужно также зафиксировать некий способ оценки качества предсказаний (именно эту оценку мы и будем в конечном счете оптимизировать). Опишем теперь в терминах TensorFlow функцию потерь. Для исходной разметки нам понадобится заглушка:

```
y_ = tf.placeholder(tf.float32, [None, 10])
```

И теперь функцию потерь тоже можно записать в одну строчку:

```
cross_entropy = tf.reduce_mean(
    -tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

Давайте более подробно разберем эту строку (в дальнейшем мы больше не будем этого делать, но один раз, пожалуй, не помешает). Посмотрим, что будет происходить, последовательно, от внутренних операций к внешним:

- сначала мы вычисляем `tf.log(y)`, логарифм каждого элемента y;
- затем умножаем каждый элемент $y\_$ на соответствующий ему `tf.log(y)` (операция умножения на векторах, матрицах и тензорах здесь понимается покомпонентно);
- затем суммируем результат с помощью `tf.reduce_sum` по второму измерению; напомним, что первое измерение — это примеры из тестового или валидационного множества, а второе — возможные классы, то есть мы суммируем не по примерам, а по размерности каждого вектора y; для этого мы указали в качестве параметра `reduction_indices=[1]`;

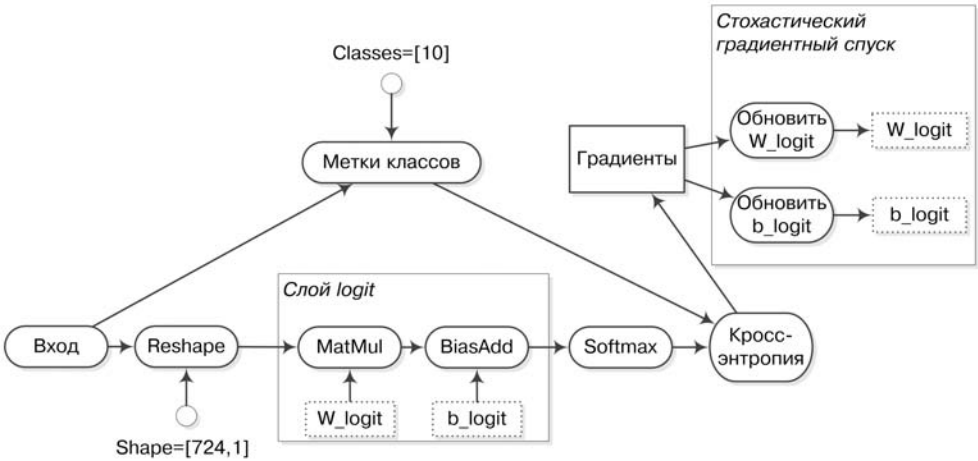


Рис. 3.10. Как проходят вычисления в модели, обучающей логистическую регрессию, при использовании библиотеки TensorFlow

- и наконец, последняя операция `tf.reduce_mean` подсчитывает среднее значение по всем примерам в выборке.

Теперь, когда мы знаем, чего мы хотим от нашей модели, мы можем попросить TensorFlow оптимизировать эту функцию. Да, это именно настолько просто. Мы уже полностью описали граф вычислений в терминах, понятных TensorFlow, все вершины в этом графе содержат известные классические функции, градиенты которых, разумеется, уже реализованы в TensorFlow, и библиотека может воспользоваться алгоритмом обратного распространения ошибки для того, чтобы подсчитать градиенты, узнать, как веса влияют на функцию потерь, которую мы хотим минимизировать, и затем применить тот или иной алгоритм оптимизации для уточнения этих весов.

Для обучения модели мы будем использовать метод градиентного спуска со скоростью обучения 0,5:

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

Иллюстрация, как известно, стоит тысячи слов, поэтому мы изобразили логику программы, которую только что написали, на рис. 3.10. Эта схема показывает, в каком порядке происходит построение модели и вычисления в ней: сначала поданное на вход изображение превращается в вектор размерности 724, затем поступает к logit-слою, который собственно вычисляет оценки нашей модели для событий, связанных с различными классами, затем softmax-функция нормализует эти оценки к вектору, похожему на вектор вероятностей. Затем он вместе с пришедшими из тренировочных данных истинными метками классов поступает на вход

функции, вычисляющей перекрестную энтропию; это и есть наша функция ошибки, от которой мы должны взять частные производные. К счастью, автоматическое дифференцирование берет на себя TensorFlow, а мы просто целомудренно скрываем происходящее в узле «Градиенты». После этого осталось только превратить значения градиентов в собственно правила пересчета весов, и все готово, можно изменять веса `w_logit` и `b_logit` на каждом шаге стохастического градиентного спуска.

Перед началом обучения осталось инициализировать переменные:

```
init = tf.initialize_global_variables()
```

Теперь мы готовы запускать обучение. Для этого создаем TensorFlow-сессию...

```
sess = tf.Session()
sess.run(init)
```

...и пора запускать!

```
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

На каждом проходе цикла мы выбираем случайные 100 примеров из обучающей выборки и передаем их в функцию `train_step` для обучения. Такой подход мы уже обсуждали — это типичный стохастический градиентный спуск; в данном случае мы применяем его потому, что обучающая выборка слишком велика для того, чтобы проходить по ней целиком на каждом шаге обучения (собственно, так будет всегда на любых данных хоть сколько-нибудь реального размера). Вместо этого мы выбираем каждый раз небольшой новый случайный набор обучающих данных и используем его.

Осталось понять, насколько хорошо мы теперь умеем распознавать рукописные цифры. Модель на данный момент выдает предсказания в виде softmax-результатов — суммирующихся в единицу чисел, отражающих уверенность модели в том или ином ответе[1]. Для того чтобы понять, какую метку мы предсказали для очередного изображения, можно просто взять максимальное значение из этих результатов. В TensorFlow это выражается как `tf.argmax`, функция, выдающая позицию максимального элемента в тензоре по заданной оси. Для того чтобы понять, верно ли мы предсказали метку, достаточно просто сравнить между собой `tf.argmax(y, 1)` и `tf.argmax(y_, 1)`:

```
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
```

Это дает на выходе список булевых значений, показывающих, верно или неверно предсказан результат; итоговой точностью может быть, например, среднее значение соответствующего вектора:

---

[1] Здесь, конечно, очень хочется просто считать результаты softmax-регрессии вероятностями классов, но мы не должны поддаваться искушению: формулу softmax-функции нелегко интерпретировать вероятностным образом, и эти значения на самом деле не очень похожи на вероятности.

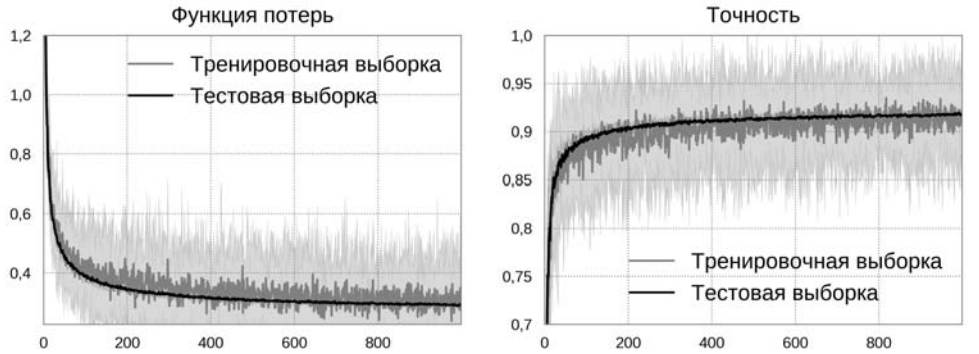


Рис. 3.11. Графики изменения функции ошибки и точности на тренировочном и тестовом множествах по мере обучения модели

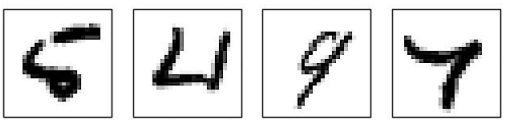


Рис. 3.12. Правильные метки — 5, 4, 9, 7. Результат классификации — 6, 6, 4, 4

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Осталось только вычислить его и вывести (на всякий случай: цифры, которые получатся у вас, наверняка будут немного отличаться от наших, они зависят от многих случайных факторов, например от случайной инициализации; но порядок величины должен быть именно такой):

```
print("Точность: %s" %
    sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
>> Точность: 0.9154
```

Поздравляем! С помощью библиотеки TensorFlow мы с вами буквально в несколько строк кода смогли построить настоящую модель для распознавания рукописных цифр, запрограммировать обучение, обучить ее на стандартном наборе данных и получить весьма неплохую точность. На рис. 3.11 изображены графики того, как функция ошибки и точность классификации меняются по мере обучения; мы запустили эксперимент 100 раз и изобразили на графиках среднее и дисперсию полученных результатов. Обратите внимание, что хотя дисперсия довольно большая, в среднем никакого оверфиттинга не происходит, точность на тестовом множестве почти точно такая же, как на тренировочном.

Кроме того, давайте ради интереса посмотрим на некоторые из тех изображений, метки которых угадать не получилось. На рис. 3.12 показаны четыре типичных изображения, на которых наш классификатор ошибается; согласитесь, случаи действительно тяжелые.

Итак, первое знакомство с TensorFlow прошло удачно, но в нашем примере не было никаких глубоких сетей, да и вообще нейронных-то сетей по сути нет: мы просто обучили логистическую регрессию.

Давайте теперь немного усложним модель и посмотрим, что из этого получится. Следующий пример можно считать, так сказать, трейлером[1] предстоящих глав: мы встретимся с некоторыми понятиями, которые более подробно разберем в дальнейших главах, а также поймем, зачем, собственно, мы все это делаем. В результате этого примера окажется, что более сложная модель действительно работает лучше; тем самым мы впервые увидим на живом примере, что строить нейронные сети все-таки имеет смысл.

Общая схема, которой будет следовать код в этом примере, изображена на рис. 3.13. В основном мы уже объяснили ее выше, когда говорили о модели логистической регрессии; что по сравнению с рис. 3.10 мы добавили еще одну группу вычислений, объединенных в «слой ReLU», а затем пропустили результат через странный узел под названием Dropout; давайте их и обсудим.

Во-первых, чтобы наша модель вообще имела право носить гордое название нейронной сети, нужно добавить в нее скрытый слой. В качестве функции активации возьмем все тот же ReLU. Нужно инициализировать веса и свободный член для этого слоя; будем в этом примере использовать 100 нейронов на скрытом слое:

```
W_relu = tf.Variable(tf.truncated_normal([784, 100], stddev=0.1))
b_relu = tf.Variable(tf.truncated_normal([100], stddev=0.1))
```

На этот раз мы инициализируем переменные не нулями, а небольшими случайными значениями. Функция `tf.truncated_normal` возвращает значения, порожденные нормально распределенной случайной величиной с фиксированными математическим ожиданием (в нашем примере мы оставили значение 0, задающееся по умолчанию) и дисперсией (в нашем примере `stddev=0.1`); однако при этом значения, вышедшие за пределы интервала в $\pm 2\sigma$ от среднего, выбираются заново, то есть распределение обрезается так, чтобы полностью запретить большие выбросы. Инициализация нулями в данном случае была бы совсем бессмысленной, потому что $\mathrm{ReLU}(0) = 0$, а значит, при инициализации нулями градиенты совсем не распространялись бы по сети. В нашем же случае примерно половина весов окажется отрицательной, и соответствующие нейроны не будут активироваться вовсе.

[1] Любопытно, что слово trailer, сейчас обозначающее рекламу предстоящих фильмов, которую показывают перед «основным блюдом», на самом деле происходит от названия прицепа сзади транспортного средства (от глагола to trail — «идти по следу»). И действительно, на заре кинематографа трейлеры показывали после фильма, но быстро поняли, что это порочная практика: их никто не смотрел.

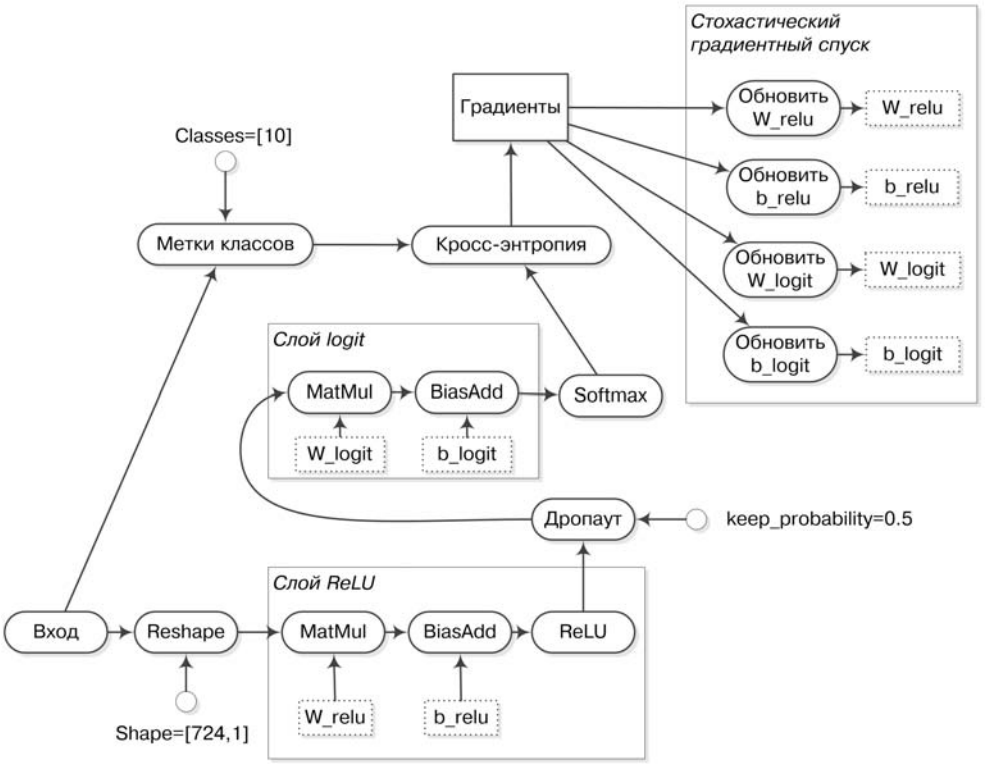


Рис. 3.13. Как проходят вычисления в нейронной сети со скрытым ReLU-слоем при использовании библиотеки TensorFlow

Итак, хотя теория градиентного спуска от этого не меняется никак, на практике хорошая инициализация весов очень важна, чтобы попасть в разумную область значений целевой функции и не застрять в плохом локальном максимуме. Мы еще поговорим о разумной инициализации весов нейронной сети в разделе 4.2, где подведем под нее и теоретические основания. А сейчас общий вид скрытого слоя получается таким:

```
h = tf.nn.relu(tf.matmul(x, W_relu) + b_relu)
```

Применение `tf.nn.relu` тоже будет покомпонентным: фактически мы просто применили функцию ReLU к вектору `tf.matmul(x, W_relu) + b_relu`.

В этой нейронной сети будет очень полезен слой *дропаута*[1]. Мы поговорим о нем подробно в разделе 4.1, а сейчас начнем с того, что предварим этот раздел

[1] Шишков, прости: мы честно пытались придумать перевод слова dropout, но ничего столь же яркого и запоминающегося не вышло, да и не встречали мы ни одного специалиста в этой области, который бы называл дропаут по-русски как-то иначе.

небольшим спойлером[1]: дропаут — это слой, который выбрасывает (обнуляет) выходы некоторых нейронов, выбираемых случайно и заново для каждого обучающего примера. Все, что нам сейчас нужно задать — это вероятность их выбрасывания; для этого мы сначала создадим заглушку:

```
keep_probability = tf.placeholder(tf.float32)
```

А дальше TensorFlow, как всегда, все делает за нас, достаточно только правильно попросить:

```
h_drop = tf.nn.dropout(h, keep_probability)
```

Теперь нейроны скрытого слоя будут участвовать в вычислениях с вероятностью `keep_probability`, а с вероятностью `1-keep_probability` их выход будет обнулен, и они не будут ни участвовать в предсказании для этого примера, ни обучаться на нем. Поскольку размер внутреннего слоя отличается от входного, нам придется немного поменять параметры внешнего слоя и, кроме того, переписать заключительный softmax-слой:

```
W = tf.Variable(tf.zeros([100, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(h_drop, W) + b)
```

При этом наша функция потерь `cross_entropy` и оптимизатор `train_step` не требуют никаких изменений. А вот вызывать `sess.run` нужно с новым параметром `keep_probability`. Поэтому цикл обучения немного изменился:

```
for i in range(2000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys,
                                    keep_probability: 0.5})
```

Если сделать 2000 шагов обучения нашей новой сети, мы получим

```
print("Точность: %s" % sess.run(accuracy, feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_probability: 1.} ))
>> Точность: 0.9657
```

Отличный результат! Мы дописали еще несколько строк кода и с самыми минимальными изменениями смогли добавить целый новый скрытый слой в модель, а также уменьшить ошибку на тестовой выборке практически в два раза. Теперь можно посмотреть, что наша новая модель делает с теми сложными примерами, с которыми обычная логистическая регрессия не справилась. В этот раз мы снова

[1] А вот за слово «спойлер» просить прощения не будем. Хотя в значении «слишком рано раскрытая сюжетная информация» оно стало употребляться недавно, само слово уже давно вошло в русский язык в достаточно широком круге значений, означающих, как и в английском, некую «помеху» в самом широком смысле: спойлер на гоночном болиде снижает сопротивление воздуха, кандидат-спойлер на выборах оттягивает голоса у другого кандидата, сам не имея шансов победить, а боксер-спойлер ведет бой вторым номером, мешает противнику вести бой и старается как можно чаще входить в клинч.

получили ту же ошибку на первом примере — вместо метки 5 модель предсказывает 6 — однако остальные три примера классифицировались правильно.

На самом деле на момент написания этой книги лучшие модели уже давно имеют точность свыше 99 %. В какой-то момент борьба шла буквально за каждую лишнюю картинку, и сейчас серьезное использование MNIST для сравнения современных моделей практически бесполезно. Исследователи обычно сравнивают результаты на других наборах данных, используя MNIST как общую проверку на осмысленность. Одну из таких более подходящих для этой задачи моделей мы рассмотрим в главе 5, когда снова вернемся к глубокому обучению на изображениях.

Но и это еще не все. Если любознательный читатель следит за нашими упражнениями с компьютером под рукой и пытается повторить то, что мы делаем, он наверняка уже попробовал увеличить длину цикла обучения, чтобы проверить: нельзя ли обучить нашу простую модель еще лучше, просто потратив на это больше времени? Но не тут-то было! Еще через 2000 шагов обучения точность неожиданно резко падает примерно до 10 %:

```
>> Точность: 0.098
```

Давайте разбираться, в чем причина. Вообще говоря, 10 % подозрительно похоже на точность генератора случайных чисел при угадывании равномерно распределенных 10 классов; поэтому первое, что нужно проверить — не сломалась ли наша модель полностью. Для этого можно, например, посмотреть на ее веса:

```
print(sess.run(b))
>> [ nan nan nan nan nan nan nan nan nan nan]
```

И действительно, элементы вектора свободных членов b перестали быть числами. Чаще всего это говорит о *переполнении*. Оно происходит, когда очень большие числа выходят за границы соответствующих диапазонов и округляются до ∞ или $-\infty$; чаще всего так получается, когда очень маленькие числа округляются до нуля, а потом на этот ноль что-нибудь пытаются разделить. Но откуда переполнение могло появиться у нас?

Одно из «узких» мест нашего кода — функция softmax (мы говорили о ней в разделе 2.3). Допустим, что все элементы вектора, который мы подаем ей на вход, одинаковы и равны некоторой постоянной c. Тогда легко видеть, что аналитически на выходе должен получаться вектор, состоящий только из $1/n$, и этот результат должен получаться совершенно независимо от c. Но численное значение может не совпасть с аналитическим! Если c будет большим отрицательным числом, знаменатель дроби может быть округлен до нуля, и мы получим неопределенность (скорее всего, с нашей моделью именно это и произошло). А если c будет большим положительным числом, то числитель может округлиться до ∞, и мы получим другую неопределенность. В случае функции softmax такие эффекты получить совсем несложно, потому что она имеет дело с экспонентами: совсем не обязательно уходить в очень глубокий минус, чтобы экспонента от отрицательного числа оказалась неотличима от нуля. Более того, в реальных моделях, обучающихся на GPU,

обычно используются числа с плавающей запятой с одинарной точностью (то есть типа `float`, а не `double`), потому что на современных GPU это получается гораздо эффективнее, а потери в качестве модели, если не наткнуться на такую вот неопределенность, совсем небольшие.

Что же делать? Оказывается, выход есть! Заметим, что если прибавить одну и ту же константу ко всем входам функции softmax, значение функции не изменится:

$$\frac{e^{x_j+c}}{\sum_i e^{x_i+c}} = \frac{e^{x_j}e^c}{\sum_i e^{x_i}e^c} = \frac{e^{x_j}e^c}{e^c\sum_i e^{x_i}} = \frac{e^{x_j}}{\sum_i e^{x_i}}.$$

Поэтому, чтобы избавиться от переполнений, достаточно вычесть из каждого входа максимум среди входов, то есть вычислять $\mathrm{softmax}(x_1,x_2,\ldots,x_n)$ так:

$$\mathrm{softmax}(x_1,x_2,\ldots,x_n) = \mathrm{softmax}(x_1 - \max_i x_i, x_2 - \max_i x_i, \ldots, x_n - \max_i x_i).$$

Тогда, с одной стороны, мы вряд ли увидим в числителе слишком большое число, а с другой, — в знаменателе всегда будет по крайней мере $e^0 = 1$, и деления на ноль точно не возникнет.

Но это не единственное место, где могла возникнуть ошибка переполнения. Давайте теперь посмотрим на нашу функцию потерь:

$$H_t(y) = -\sum_i t_i \log y_i.$$

Когда мы вычисляем ее градиент по весам модели, нам нужно будет подсчитать, в частности,

$$\frac{\partial H}{\partial y_i} = -\frac{t_i}{y_i}.$$

Тогда, если предсказанная моделью оценка для какого-то класса окажется очень мала (то есть модель уверена в том, что пример этому классу совсем не принадлежит), то мы снова будем пытаться разделить на ноль и получим неопределенность при $t_i = 0$ или бесконечность в случае $t_i = 1$.

К счастью, создатели TensorFlow знали об этих трудностях и специально подготовили функцию, которая обходит все подводные камни. Давайте использовать ее вместо самописных функций. Зададим промежуточный тензор

```
logit = tf.matmul(h_drop, W) + b
```

Теперь мы не будем сами применять softmax или считать перекрестную энтропию, а воспользуемся готовой функцией:

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logit, y_))
```

Больше ничего менять не надо; давайте снова запустим код, теперь уже сделав 10 000 шагов обучения, и посмотрим на результат:

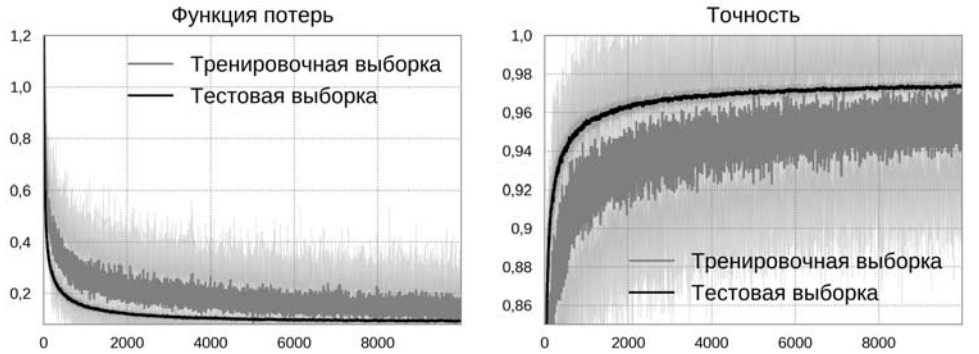


Рис. 3.14. Графики изменения функции ошибки и точности на тренировочном и тестовом множествах по мере обучения модели

```
for i in range(10000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={
        x: batch_xs, y_: batch_ys, keep_probability: 0.5})
print("Точность: %s" %
    sess.run(accuracy, feed_dict={
        x: mnist.test.images, y_: mnist.test.labels, keep_probability: 1.}))
>> Точность: 0.9749
```

Неплохо! Уже 97,5 % — теперь наша модель ошибается (на тестовой выборке, естественно) всего один раз из сорока. Графики изменения функции ошибки и точности классификации на обучающей и тестовой выборках показаны на рис. 3.14; на этот раз мы показали 10 000 итераций, потому что в течение всего этого времени точность на тестовом множестве продолжала увеличиваться. Обратите внимание, что качество на тестовом множестве стабильно выше, а дисперсия ниже, чем на тренировочном — это эффект дропаута: мы не пересчитывали отдельно качество на тренировочном множестве, а брали его из процесса обучения, с дропаутом.

Итак, в этой главе мы подробно рассмотрели процедуру автоматического дифференцирования на графе вычислений, а в данном разделе познакомились с тем, как описывать и обучать модели в TensorFlow. Оказывается, благодаря автоматическому дифференцированию можно начинать обучать весьма разумные модели, в том числе глубокие нейронные сети, буквально за несколько строк кода, почти совсем не задумываясь о том, как же вычислять в них градиент. В следующих главах мы познакомимся с чуть более сложными архитектурами нейронных сетей, которые часто применяются на практике, и еще не раз убедимся в том, насколько удобно пользоваться библиотеками вроде TensorFlow или Theano.

Часть II

Основные архитектуры

Глава 4
Быстрее, глубже, сильнее,
или Об оврагах, долинах и трамплинах

TL;DR

Четвертая глава посвящена прогрессу в методах оптимизации и регуляризации нейронных сетей. Прогресс этот на первый взгляд незаметен, но во многом определяет успех современных нейронных сетей. Мы рассмотрим:

- новейшие методы регуляризации, которые от сокращения весов перешли к дропауту и другим подходам;
- современные методы случайной инициализации весов, которые позволяют контролировать дисперсию выходов;
- метод нормализации по мини-батчам, который позволяет не обучать веса заново при изменении предыдущих слоев;
- улучшения градиентного спуска с помощью метода моментов;
- и даже адаптивные варианты градиентного спуска.

4.1. Регуляризация в нейронных сетях

> Тем не менее изображенное на нем ничуть не отвечало современности ни по духу, ни по замыслу, поскольку, при всей причудливости и разнообразии кубизма и футуризма, они редко воспроизводят ту загадочную регулярность, которая таится в доисторических письменах.
>
> *Г. Лавкрафт. Зов Ктулху*

Эта глава будет посвящена тем видам прогресса в обучении нейронных сетей, которые на первый взгляд кажутся незаметными. В самом деле, в разделе 2.5 мы говорили о том, что если мы можем определить структуру сети, то дальше задача как бы уже и решена, сведена к «всего лишь оптимизации», которую можно делать градиентным спуском, ведь мы же умеем брать производную от одного перцептрона, правильно? Таким образом, получается, что прогресс в области нейронных сетей должен выглядеть так: придумали новую архитектуру, засунули в «универсальный оптимизатор», в роли которого может выступить TensorFlow, посмотрели, не стало ли лучше, придумали еще одну архитектуру, снова быстренько реализовали на TensorFlow...

И такой прогресс, конечно, есть. Многие главы этой книги будут посвящены именно разным архитектурам современных нейронных сетей. Однако в вышеупомянутой «всего лишь оптимизации» тоже кроется немало интересного. В настоящей главе мы поговорим о том, какие «трюки» позволяют нам обучать такие гигантские модели, как современные нейронные сети. Многие из этих трюков были придуманы совсем недавно, и можно сказать, что они тоже отделяют современную революцию нейронных сетей от того, как люди пытались обучать сети раньше. Мы уже говорили о том, что поначалу глубокие сети научились обучать с помощью достаточно сложных моделей для предобучения без учителя, а потом оно снова стало не обязательным, — так вот, в этой главе мы узнаем, почему оно теперь больше не нужно.

Первая важная тема — регуляризация в нейронных сетях. В разделе 2.2 мы видели, что модель, у которой слишком много свободных параметров, плохо обобщается, то есть слишком близко «облизывает» точки из тренировочного множества и в результате недостаточно хорошо предсказывает нужные значения в новых точках. Современные нейронные сети — это модели с огромным числом параметров, даже не самая сложная архитектура может содержать миллионы весов. Значит, надо регуляризовать.

Первая идея: давайте воспользуемся теми же методами, которые разрабатывали в разделе 2.2 — потребуем от каждой переменной, которую мы оптимизируем, принимать не слишком большие значения. Это можно сделать так же, как мы делали тогда, добавив к целевой функции регуляризаторы в любом удобном для вас виде. Обычно используют два их вида:

- *L_2-регуляризатор*, сумму квадратов весов $\lambda \sum_w w^2$;
- *L_1-регуляризатор*, сумму модулей весов $\lambda \sum_w |w|$.

Разумеется, оптимизация от этого не сильно пострадает, ведь от регуляризатора тоже легко взять производную; просто немного изменится целевая функция.

В теории нейронных сетей такая регуляризация называется *сокращением весов* (weight decay), потому что действительно приводит к уменьшению их абсолютных значений. Его, конечно, применяли очень давно, в 1980-х годах уж точно [207, 285]. И сейчас в таких библиотеках, как TensorFlow и Keras, вы тоже можете очень легко применить сокращение весов. Например, в Keras есть возможность для каждого слоя добавить регуляризатор на три вида связей:

- `kernel_regularizer` — на матрицу весов слоя;
- `bias_regularizer` — на вектор свободных членов;
- `activity_regularizer` — на вектор выходов.

Примерно так:

```
model.add(Dense(256, input_dim=32,
          kernel_regularizer=regularizers.l1(0.001),
          bias_regularizer=regularizers.l2(0.1),
          activity_regularizer=regularizers.l2(0.01)))
```

Регуляризация в форме сокращения весов применяется до сих пор, однако сильно увлекаться подбором регуляризаторов мы не советуем.

Во-первых, есть еще один метод регуляризации, совсем простой и очевидный: давайте просто отложим часть тренировочного набора (назовем отложенную часть *валидационным множеством*) и будем при обучении на основном тренировочном множестве заодно вычислять ошибку и на валидационном. Предположение здесь в том, что ошибка на валидационном множестве будет хорошо оценивать ошибку и на новых точках (тестовом множестве), ведь она взята из данных той же природы, но тех, на которых мы не обучались. И остановить обучение нужно будет не тогда, когда сеть придет в локальный оптимум для тренировочного множества, а тогда, когда начнет ухудшаться ошибка на валидационном множестве. В теории нейронных сетей этот подход обычно называется методом *ранней остановки* (early stopping). Есть результаты о том, что ранняя остановка в некотором смысле эквивалентна L_2-регуляризации [88]. В целом, вне зависимости от того, как именно вы будете делать раннюю остановку, мы в любом случае всецело рекомендуем отложить валидационную часть датасета и следить за ошибкой на этой части — это всегда полезно и дает хорошую оценку способности модели к обобщению. Если, конечно, размер датасета позволяет такую расточительность.

Во-вторых, в последние годы были разработаны более эффективные методы, которые стоит применить в первую очередь. При этом они часто работают настолько хорошо, что до «обычного» сокращения весов дело может в итоге и не дойти. Одним из важнейших методов регуляризации нейронных сетей для революции глубокого обучения стал *дропаут* [129, 246].

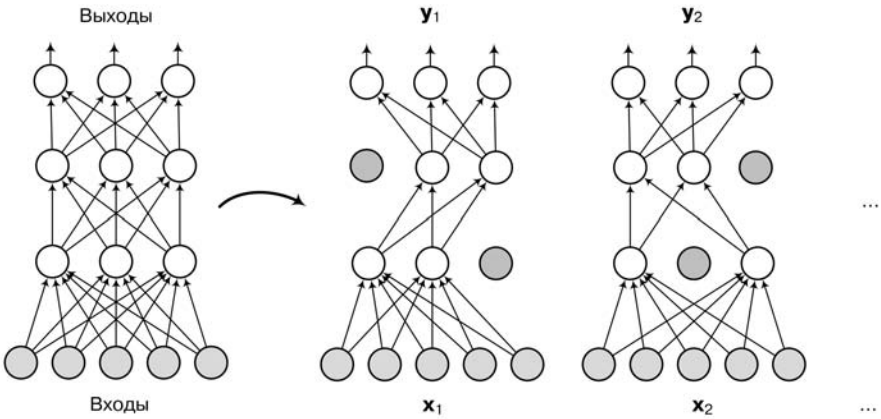


Рис. 4.1. Пример дропаута в трехслойной нейронной сети: на каждом новом тренировочном примере структура сети изменяется

Его идея чрезвычайно проста. Давайте для каждого нейрона (кроме самого последнего, выходного слоя) установим некоторую вероятность p, с которой он будет... *выброшен из сети*. Алгоритм обучения меняется таким образом: на каждом новом тренировочном примере $\boldsymbol{x}$ мы сначала для каждого нейрона бросаем монетку с вероятностью p, и в зависимости от результата монетки либо используем нейрон как обычно, либо устанавливаем его выход всегда строго равным нулю. Дальше все происходит без изменений; ноль на выходе приводит к тому, что нейрон фактически выпадает из графа вычислений: и прямое вычисление, и обратное распространение градиента останавливаются на этом нейроне и дальше не идут.

Пример дропаута показан на рис. 4.1, где изображена трехслойная нейронная сеть с пятью входами и тремя слоями по три нейрона. На выходном слое дропаут обычно не делают: нам требуется выход определенной размерности, и все его компоненты обычно нужны. А на промежуточных, скрытых слоях дропаут можно применить: на рис. 4.1 видно, как каждый новый тренировочный пример $\boldsymbol{x}_i$ обучает уже немножко другую сеть, где часть соединений выброшена.

Вот и все! Очень простая идея и никакой математики. Основной математический результат работ [129, 246] состоит в ответе на вопрос о том, как же потом *применять* обученную сеть. На первый взгляд кажется, что нужно опять сэмплировать кучу «прореженных» сетей, считать их результаты, усреднять... нет, это не будет работать. Но на самом деле большой вычислительной сложности здесь нет: усреднение будет эквивалентно применению сети, в которой никакие нейроны не выброшены, но выход каждого нейрона умножен на вероятность p, с которой нейрон оставляли при обучении. Математическое ожидание выхода нейрона при этом сохранится, а это именно то, что нужно. Кстати, практика показывает, что для очень широкого спектра архитектур и приложений замечательно подходит $p = \frac{1}{2}$, так что эту вероятность можно специально и не подбирать.

И эксперименты в [129, 246], и вся последующая практика обучения нейронных сетей показывают, что дропаут действительно дает очень серьезные улучшения в качестве обученной модели в самых разных приложениях. Мы с вами тоже уже применяли его на практике в разделе 3.6, так что не будем повторяться. Однако история еще не закончена, ведь на первый взгляд дропаут выглядит ужасно загадочно: почему вдруг надо выкидывать нейроны? Какой в этом смысл?

Одной из первых попыток объяснить это явление концептуально были рассуждения Джеффри Хинтона в исходной статье о дропауте [129] и сопутствующих докладах. Хинтон полагал, что дропаут — это своеобразный метод добиться огромного усреднения моделей. Известно, что это очень полезный на практике трюк; многие Kaggle-соревнования[1] выигрываются при помощи большого ансамбля моделей[2]. И Хинтону с соавторами удалось показать, что дропаут эквивалентен усреднению всех тех моделей, которые получались на каждом шаге случайным выбрасыванием отдельных нейронов. Иначе говоря, мы усредняем 2^N возможных моделей (где N — число нейронов, которые могут быть выкинуты или оставлены), да не просто разных моделей, а разных *архитектур* нейронных сетей!

Другая мотивация для дропаута приходит из теории эволюции: методика дропаута очень похожа на *половое размножение*. Действительно, на первый взгляд кажется, что эволюции нет нужды придумывать разнополых существ: если какие-то гены научатся хорошо «работать вместе», то лучше передать эту хорошую комбинацию дальше, в то время как половое размножение может нарушить адаптивные сочетания генов.

Однако выясняется, что на самом деле такое разрушение для эволюции скорее полезно: все самые высокоорганизованные животные размножаются именно половым путем, а почкование — удел губок и кишечнополостных. Объяснение этому примерно то же, что и объяснение полезности дропаута: важно не столько собрать хорошую комбинацию генов, сколько собрать *устойчивую* и хорошую комбинацию генов, которая потом будет широко воспроизводиться и сможет стать основой для новой линии потомков. А этого проще достичь, если заставлять гены, как признаки в нейронной сети, «работать» поодиночке, не рассчитывая на соседа (который при половом размножении может просто пропасть)[3].

---

[1] Kaggle — это широко известный в среде анализа данных сайт, на котором публикуются условия различных соревнований: суть задачи, тренировочный набор, скрытый от участников тестовый набор, на котором будут оцениваться результаты, а также зачастую финансовая мотивация для победителей. Для людей, изучающих анализ данных, Kaggle — отличный способ попрактиковаться, а для компаний — не менее отличный способ заставить массу умных и опытных людей думать над своей задачей, потратив на это считанные единицы тысяч долларов в качестве приза.

[2] Можно, конечно, построить и настоящий ансамбль из нейронных сетей; правда, если делать это наивным образом, буквально обучая много сетей, это будет очень сложно вычислительно и вряд ли приведет к успеху. Но в последнее время стали появляться интересные работы о том, как здесь тоже можно сэкономить и задешево обучить настоящий ансамбль [500].

[3] Желающим подробнее узнать, как информатика смотрит на половое размножение, рекомендуем работы [331, 489].

И наконец, третий возможный взгляд: дропаут очень похож на то, что происходит при связях между живыми нейронами в человеческом мозге. Мы уже говорили, что нейроны работают стохастически, то есть посылают сигналы не постоянно, а в виде случайного процесса, и его интенсивность зависит от того, возбужден нейрон или нет. Взгляды эти, конечно, очень интересные и наверняка верные, но сложно понять, какие из них можно сделать выводы: и так было понятно, что дропаут полезен, и делать его надо. Однако недавно был разработан другой, более математически концептуальный взгляд на дропаут, из которого вытекают новые интересные факты; в частности, он привел к разработке правильного метода того, как делать дропаут в рекуррентных сетях. Сейчас нам, правда, еще рановато о нем говорить; давайте отложим этот разговор до конца книги, до раздела 10.5, а сами пойдем дальше — к инициализации весов.

4.2. Как инициализировать веса

> Я хочу стать художником... Что же мне нарисовать? Нарисую петушка! Я их много видел — и живых петушков и нарисованных... Вот только с чего начать? С головы или с хвоста? Самое трудное — это начать! Начну с головы, а потом подрисую все остальное: хвост, крылья и лапки со шпорами... Вот! Хороший получился гребешок. Как настоящий! Теперь нарисую глаза и клюв... Что такое? Получился попугай!
>
> *С. Михалков. С чего начать?*

В этом разделе мы обсудим еще одну проблему, которая оказывается особенно важной для глубоких нейронных сетей. Как мы уже обсуждали, обучение сети — это большая и сложная задача оптимизации в пространстве очень высокой размерности. Решаем мы ее фактически методами локального поиска: сколько ни придумывай хитрых способов ускорить градиентный спуск, обойти небольшие локальные минимумы, выбраться из «ущелий», мы все равно не сможем изменить тот факт, что градиентный спуск — это метод местного значения, и ищет он только локальный минимум/максимум. Мы не знаем никаких методов глобальной оптимизации, которые позволили бы найти самый лучший локальный оптимум для такой сложной задачи, как обучение глубокой нейронной сети. Поэтому вполне естественно, что один из ключевых вопросов состоит в том, *где начинать* этот локальный поиск: в зависимости от качества начального приближения можно попасть в самые разные локальные оптимумы. Хорошая инициализация весов может позволить нам обучать глубокие сети и лучше (в смысле метрик качества), и быстрее (в смысле числа требующихся обновлений весов, то есть числа итераций, то есть времени обучения).

Мы уже упоминали первую идею, которая привела к большим успехам в этом направлении: *предобучение без учителя* (unsupervised pretraining). Можно обучать

отдельные слои глубокой сети без учителя, последовательно, а затем веса полученных слоев считать начальным приближением и дообучать уже на размеченном наборе данных. Как это часто бывает в исследованиях нейронных сетей, эта идея тоже появилась еще во второй половине 1980-х годов: в опубликованной в 1986 году работе [26] строится глубокая архитектура, отдельные части которой обучаются как автокодировщики[1], а затем объединяются в общую модель.

Однако настоящий расцвет предобучения без учителя начался в середине первого десятилетия XXI века. Именно оно обусловило ту самую революцию обучения глубоких сетей, которой вся эта книга обязана своим появлением. Любопытно, что хотя сейчас обучение нейронных сетей можно назвать достаточно простым для изучения инструментом[2], и в этой книге мы достаточно быстро объясняем обучение нейронных сетей, начиная от самых азов, в развитии метода без вероятностных моделей и методов приближенного вероятностного вывода не обошлось. Основным инструментом для предобучения без учителя в работах группы Хинтона стали так называемые *ограниченные машины Больцмана* (restricted Boltzmann machines).

В этой книге мы не будем подробно объяснять, что это такое. Достаточно будет сказать, что это вероятностная модель, также известная с 80-х годов прошлого века [2, 225, 499], в которой есть *видимые* (visible) переменные, значения которых мы знаем, и *скрытые* (hidden), значений которых мы не знаем, а также связи между ними. И задача состоит в том, чтобы обучить веса этих связей так, чтобы распределение, порождаемое скрытыми переменными на видимых, было как можно больше похоже на распределение входных данных.

Алгоритм, который такое обучение реализует, получил название *contrastive divergence*[3]. Он представляет собой упрощенную версию алгоритма сэмплирования Монте-Карло для соответствующей вероятностной модели. Этот алгоритм тоже придуман Хинтоном, причем еще в 2002 году, до начала революции глубокого обучения [219], а в 2005 году в работе [72] появилась ключевая идея предобучения: начать с contrastive divergence, чтобы обучить начальную точку, а потом дообучить результат при помощи обычного обучения с учителем. Таким же образом можно построить и глубокую модель: сначала обучить первый слой как ограниченную

---

[1] Мы подробно поговорим об автокодировщиках в главе 5, а сейчас достаточно сказать, что автокодировщик — это модель, которая обучает без учителя некоторое внутреннее представление своих входов и умеет их затем реконструировать.

[2] По сравнению с другими, конечно же! Но поверьте, математическое содержание всей этой книги, кроме, возможно, главы 10, достаточно несложно и недалеко уходит от базовых университетских курсов теории вероятностей и математического анализа. С другой стороны, не надо путать простоту в освоении с простотой для понимания: люди до сих пор не очень понимают, *почему* так хорошо работают те методы, о которых мы рассказываем в этой книге, и в наших представлениях о том, что происходит в сложных нейронных сетях, еще очень, очень много пробелов.

[3] Еще один термин, перевод которого на русский не устоялся и, по всей видимости, уже и не устоится. Мы встречали «сравнительное расхождение», «сопоставительное отклонение» и даже «контрастную дивергенцию», но обычно так и пишут простыми латинскими буквами: «алгоритм contrastive divergence», сокращенно CD.

машину Больцмана, затем использовать скрытые вершины первого слоя как видимый слой второго уровня, потом так же с третьим... а в конце объединить все в одну большую модель с несколькими слоями, использовать веса машин Больцмана как начальное приближение и дообучить результат с учителем.

Такая конструкция получила название *глубокой машины Больцмана* (Deep Boltzmann machine, DBN) [466]. После своего появления в 2005–2006 годах этот подход к предобучению несколько лет применялся очень активно; он подробно описан в уже ставших классическими публикациях Джеффри Хинтона и его коллег и аспирантов, особенно Руслана Салахутдинова [220, 221, 224, 464, 467, 468, 505, 516]. Есть и теоретические результаты, показывающие, что вся эта конструкция действительно делает то, что надо, то есть улучшает некоторую оценку общего правдоподобия модели [223].

Можно подойти к этому вопросу и с другой стороны. Ограниченная машина Больцмана обучает веса между видимым и скрытым слоем так, чтобы как можно лучше по скрытому слою восстанавливать видимый... ничего не напоминает? Действительно, это весьма похоже на ситуацию обучения одного слоя нейронной сети, если еще точнее — на обучение однослойного автокодировщика. И действительно, вскоре появились и аналогичные глубокие конструкции, основанные на автокодировщиках: обучили первый автокодировщик, использовали выделяемые им признаки как входы второго и т. д. Этот подход, известный как *многоярусные автокодировщики* (stacked autoencoders), развивался в тот же период в основном в группе Йошуа Бенджи [148, 195], а в контексте сверточных сетей — в группе Яна ЛеКуна [303].

Почему это работает? Масштабное исследование [567], в 2010 году предпринятое в группе Бенджи, показало, что предобучение без учителя действительно помогает практически любой глубокой модели, и выдвинуло интересную гипотезу о том, почему так происходит. Авторы предположили, что предобучение выступает в качестве своеобразного метода регуляризации, связанного с общим подходом к вероятностным моделям. Дело в том, что в то время как «обычное» обучение глубокой сети касается только восстановления выхода $\boldsymbol{y}$ по входу $\boldsymbol{x}$, то есть пытается обучить *условное* распределение $p(\boldsymbol{y} \mid \boldsymbol{x})$, порождающая вероятностная модель (такая, как, например, машина Больцмана) обучает *совместное* распределение $p(\boldsymbol{x},\boldsymbol{y}) = p(\boldsymbol{y} \mid \boldsymbol{x})p(\boldsymbol{x})$, то есть старается еще и выразить распределение данных, которые попадаются на входах сети (мы подробнее поговорим о порождающих моделях в разделе 8.2). В итоге это приводит к инициализации сети в той области пространства весов, где веса описывают и условное распределение $p(\boldsymbol{y} \mid \boldsymbol{x})$, и собственно распределение данных $p(\boldsymbol{x})$, и в результате есть шанс, что обучение с учителем затем сойдется в более хорошо обобщающийся и менее склонный к оверфиттингу локальный оптимум, чем при случайной инициализации[1].

[1] Кстати, недавние работы [317, 487, 566] показывают, что предобучение можно заменить и даже улучшить, обучаясь сразу на размеченных и неразмеченных данных при помощи методов обучения

Какой бы способ предобучения ни использовался, в подавляющем большинстве случаев он соответствует ряду достаточно естественных принципов.

1. Предобучение слоев происходит последовательно, от нижних к верхним. Это позволяет избежать проблемы затухающих градиентов и существенно уменьшает объем вычислений на каждом этапе.
2. Предобучение протекает без учителя, то есть без учета имеющихся размеченных данных. Это часто позволяет существенно расширить обучающую выборку: понятно, что собрать миллионы изображений из Интернета без учета контекста и описания куда проще, чем собрать даже тысячу правильно размеченных рукописных цифр, а собрать много сырой человеческой речи куда проще, чем речь с размеченными фонемами.
3. В результате предобучения получается модель, которую затем нужно дообучить на размеченных данных. И модели, обученные таким образом, в конечном счете стабильно сходятся к существенно лучшим решениям, чем при случайной инициализации.

...По крайней мере, так было во второй половине 2000-х годов, когда предобучение посредством ограниченных машин Больцмана или автокодировщиков было в расцвете. Сейчас его тоже никто не мешает делать, скорее всего, результаты слегка улучшатся. Но все-таки предобучение — это достаточно сложная и дорогостоящая процедура. Фактически получается, что нужно научиться делать две разных процедуры обучения, настраивать их обе, причем предобучение скорее всего будет более хрупкой и сложной фазой, чем собственно обучение с учителем. Конечно, хотелось бы как-нибудь обойтись без предобучения, так что кроме вопроса о том, почему оно работает хорошо, люди задались и обратным вопросом: почему случайная инициализация работает плохо и можно ли что-то сделать для того, чтобы ее улучшить?

Важной вехой на этом пути стала совместная работа Ксавье Глоро (Xavier Glorot) и Йошуа Бенджи [179], вышедшая в 2010 году. Формально поставленный там вопрос был именно «исследовательский»: почему глубокие сети трудно обучать сразу целиком? Выяснилось, что результат обучения на практике очень сильно зависит от первоначальных значений весов. И хотя полноценное предобучение с помощью ограниченных машин Больцмана в современной практике встречается редко, все равно очень важно инициализировать веса правильно. И основным результатом [179] стал простой и хорошо мотивированный способ инициализации весов, позволяющий существенно ускорить обучение и улучшить качество. В результате он даже получил название в честь Глоро: его часто называют *инициализацией Ксавье* (Xavier initialization).

с частичным привлечением учителя (semi-supervised learning). Более того, модели, оптимизирующие функцию ошибки, которая комбинирует в себе как ошибку на правильных ответах (supervised), так и часть, отвечающую за восстановление исходных данных (unsupervised), обычно показывают результаты лучше, чем при простом предобучении. Этот подход пока не вошел в мейнстрим, но уж сноски точно заслуживает...

Для того чтобы понять, в чем идея инициализации Ксавье, давайте рассмотрим дисперсию значений активации у слоев нейронной сети. Для начала покажем, как связаны дисперсии последовательных слоев сети. У одного-единственного нейрона значение активации (то есть выход перед функцией активации) выглядит так:

$$y = \boldsymbol{w}^\top \boldsymbol{x} + b = \sum_i w_i x_i + b,$$

где $\boldsymbol{x}$ — вектор входных значений, а $\boldsymbol{w}$ — вектор весов нейрона. Получается, что дисперсия $\mathrm{Var}(y)$ не зависит от свободного члена b и выражается через дисперсии $\boldsymbol{x}$ и $\boldsymbol{w}$. Для i-го слагаемого суммы $\sum_i w_i x_i$, которое мы обозначим как $y_i = w_i x_i$, в предположении о том, что w_i и x_i независимы (что вполне естественно), мы получим дисперсию:

$$\begin{aligned}\mathrm{Var}\left(y_i\right) = \mathrm{Var}\left(w_i x_i\right) = \mathbb{E}\left[x_i^2 w_i^2\right] - \left(\mathbb{E}\left[x_i w_i\right]\right)^2 = \\ = \mathbb{E}\left[x_i\right]^2 \mathrm{Var}\left(w_i\right) + \mathbb{E}\left[w_i\right]^2 \mathrm{Var}\left(x_i\right) + \mathrm{Var}\left(w_i\right) \mathrm{Var}\left(x_i\right).\end{aligned}$$

Если мы используем симметричные функции активации и случайно инициализируем веса со средним значением, равным нулю, то первые два члена последнего выражения оказываются тоже равными нулю, а значит:

$$\mathrm{Var}\left(y_i\right) = \mathrm{Var}\left(w_i\right) \mathrm{Var}\left(x_i\right).$$

Если теперь мы предполагаем, что как x_i, так и w_i инициализируются из одного и того же распределения, причем независимо друг от друга (это сильное предположение, но в данном случае вполне естественное), мы получим:

$$\mathrm{Var}\left(y\right) = \mathrm{Var}\left(\sum_{i=1}^{n_{\text{out}}} y_i\right) = \sum_{i=1}^{n_{\text{out}}} \mathrm{Var}\left(w_i x_i\right) = n_{\text{out}} \mathrm{Var}\left(w_i\right) \mathrm{Var}\left(x_i\right),$$

где n_{out} — число нейронов последнего слоя. Другими словами, дисперсия выходов пропорциональна дисперсии входов с коэффициентом $n_{\text{out}} \mathrm{Var}\left(w_i\right)$.

До работы [179] стандартным эвристическим способом случайно инициализировать веса новой сети[1] было равномерное распределение следующего вида:

$$w_i \sim U\left[-\frac{1}{\sqrt{n_{\text{out}}}}, \frac{1}{\sqrt{n_{\text{out}}}}\right].$$

[1] См., например, классическую работу ЛеКуна [136], которая была опубликована в 1998 году в книге *Neural Networks: Tricks of the Trade* [388], содержащей практические советы по обучению нейронных сетей. Кстати, в 2012 году вышло второе издание книги [389], которое уже вполне современно, и мы его всецело рекомендуем.

В этом случае получается, что:

$$\mathrm{Var}\,(w_i) = \frac{1}{12}\left(\frac{1}{\sqrt{n_{\text{out}}}} + \frac{1}{\sqrt{n_{\text{out}}}}\right)^2 = \frac{1}{3n_{\text{out}}}, \quad \text{и} \quad n_{\text{out}}\mathrm{Var}\,(w_i) = \frac{1}{3}.$$

После нескольких слоев такое преобразование параметров распределения значений между слоями сети фактически приводит к затуханию сигнала: дисперсия результата слоя каждый раз уменьшается, фактически делится на 3, а среднее у него было нулевое.

Аналогичная ситуация повторяется и на шаге обратного распространения ошибки при обучении. Например, если $z^{(l+1)} = f\left(y^{(l)}\right)$, где l — номер слоя, а f — функция активации, то производная функции ошибки L, согласно обычной процедуре обратного распространения ошибки, будет такой:

$$\frac{\partial L}{\partial y_i^{(l)}} = f'(y_i^{(l)}) \sum_j w_{i,j}^{(l+1)} \frac{\partial L}{\partial y_j^{(l+1)}}.$$

Если мы используем симметричную функцию активации с единичной производной в окрестности нуля (например, tanh), то $f'(y_i^{(l)}) \approx 1$, и наблюдается ситуация, аналогичная вычислению функции, но теперь мы получим коэффициент пропорциональности для дисперсии $n_{\text{in}}\mathrm{Var}\,(w_i)$, где n_{in} — число нейронов во входном слое, а не на выходе.

Идея Глоро и Бенджи заключается в том, что для беспрепятственного распространения значений активации и градиента по сети дисперсия в обоих случаях должна быть примерно равна единице. Поскольку для неодинаковых размеров слоев невозможно удовлетворить оба условия одновременно, они предложили инициализировать веса очередного слоя сети симметричным распределением с такой дисперсией:

$$\mathrm{Var}\,(w_i) = \frac{2}{n_{\text{in}} + n_{\text{out}}},$$

что для равномерной инициализации приводит к следующему распределению:

$$w_i \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}\right].$$

Давайте поставим эксперимент и проверим, так ли хороша на самом деле эта нормализованная инициализация. Для этого зададим простую полносвязную модель и будем оценивать точность на тестовом множестве в датасете MNIST. Сначала импортируем все необходимое из Keras:

```
from keras.models import Sequential
from keras.layers import Dense
```

Как и в TensorFlow, в Keras набор данных MNIST доступен «из коробки»:

```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Однако на этот раз правильные ответы заданы в виде цифр, и нам придется самостоятельно перекодировать их в виде векторов. Для этого можно использовать модуль `np_utils`, входящий в состав Keras:

```
from keras.utils import np_utils
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

Теперь осталось для удобства перевести матрицы `X_train` и `X_test` из целочисленных значений на отрезке $[0, 255]$ к вещественным на $[0, 1]$ (нормализовать), а также сделать из квадратных изображений размера 28×28 пикселов одномерные векторы длины 784; это значит, что сами тензоры `X_train` и `X_test` будут иметь размерность (число примеров) $\times$ 784:

```
X_train = X_train.reshape([-1, 28*28]) / 255.
X_test = X_test.reshape([-1, 28*28]) / 255.
```

Все, данные готовы. Поскольку мы собираемся определять сразу две одинаковые модели, различающиеся только способом инициализации весов, давайте сразу объявим соответствующую функцию.

```
def create_model(init):
    model = Sequential()
    model.add(Dense(100, input_shape=(28*28,), init=init, activation='tanh'))
    model.add(Dense(100, init=init, activation='tanh'))
    model.add(Dense(100, init=init, activation='tanh'))
    model.add(Dense(100, init=init, activation='tanh'))
    model.add(Dense(10, init=init, activation='softmax'))
    return model
```

В этом коде функция `create_model` принимает на вход текстовый параметр, который интерпретируется как тип инициализации. Для нашего эксперимента это будут значения `uniform` и `glorot_normal`. А возвращает функция простую полносвязную модель с четырьмя промежуточными слоями, каждый размера 100. Везде, кроме последнего слоя, мы используем в качестве функции активации гиперболический тангенс, а в последнем слое — softmax, так как собираемся использовать в качестве функции потерь перекрестную энтропию.

Процесс компиляции модели и ее обучения задается очень просто:

```
uniform_model = create_model("uniform")
uniform_model.compile(
    loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
uniform_model.fit(x_train, Y_train,
    batch_size=64, nb_epoch=30, verbose=1, validation_data=(x_test, Y_test))
```

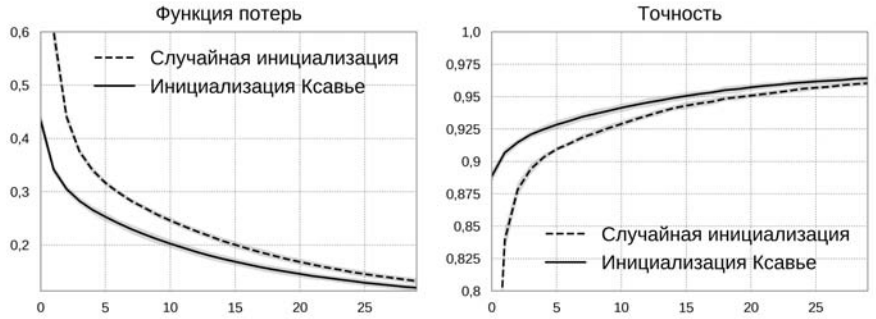


Рис. 4.2. Сравнение инициализации Ксавье и случайной инициализации весов

```
glorot_model = create_model("glorot_normal")
glorot_model.compile(
    loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
glorot_model.fit(x_train, Y_train,
    batch_size=64, nb_epoch=30, verbose=1, validation_data=(x_test, Y_test))
```

Для того чтобы достаточно «честно» и детально сравнить два способа инициализации, мы обучали каждую из моделей в течение 30 эпох по 10 раз каждую, а затем подсчитали среднее значение точности и ее дисперсию после каждой эпохи. Результат показан на рис. 4.2, где мы изобразили как среднее (собственно кривые), так и дисперсию значений ошибки и точности (затемненная область вокруг кривых показывает доверительный интервал в две дисперсии от среднего). Видно, что при инициализации весов по методу Ксавье модель уже после первой эпохи находит решение с точностью около 90 %, на что модели, чьи веса инициализированы равномерным распределением, требуется около 10 эпох. И при дальнейшем обучении «равномерная» модель все время продолжает проигрывать модели с инициализацией Ксавье. Более того, и результаты, и улучшение получаются в высшей степени устойчивыми.

Кстати, в работе [179] был сделан и еще один очень интересный и практически важный вывод. Оказывается, логистический сигмоид σ — это весьма неудачная функция активации для глубоких нейронных сетей! Эксперименты Глоро и Бенджи с глубокими сетями, основанными на логистическом сигмоиде, показали поведение, полностью соответствующее тому, о котором мы уже рассуждали в разделе 3.5: последний уровень сети очень быстро насыщается, и выбраться из этой ситуации насыщения глубокой сети очень сложно. Почему так?

Хотя доказать строгое утверждение о том, что здесь происходит, довольно сложно, есть правдоподобное объяснение, которое как раз различает логистический сигмоид и другие симметричные функции активации, например

гиперболический тангенс tanh. Рассмотрим последний слой сети, который выглядит как $f(W\boldsymbol{h}+\boldsymbol{b})$, где $\boldsymbol{h}$ — выходы предыдущего слоя, $\boldsymbol{b}$ — свободные члены последнего уровня, а f — функция активации последнего уровня, обычно softmax. Когда мы начинаем оптимизировать сложную функцию потерь по параметрам нейронной сети и оцениваем качество как среднюю ошибку на тестовом множестве, поначалу выходы $\boldsymbol{h}$ фактически не несут никакой полезной информации о входах, ведь первые уровни еще совсем не обучены.

В такой ситуации оказывается, что неплохим приближением к оптимальному результату может служить константная функция, выдающая средние значения выходов. Это значит, что значение активации выходного слоя сети $f(W\boldsymbol{h}+\boldsymbol{b})$ будет обучаться так: сеть подберет подходящие свободные члены $\boldsymbol{b}$ и постарается обнулить слагаемое $W\boldsymbol{h}$, которое на ранних этапах обучения выступает в роли скорее шума, чем полезного сигнала.

Иначе говоря, в процессе обучения мы постараемся привести выходы предыдущего слоя к нулю. И тут-то и проявляется разница: функция $\sigma(x)=\frac{1}{1+e^{-x}}$ имеет область значений $(0,1)$ при среднем значении $\frac{1}{2}$, и если мы попытаемся приблизить ее значение к нулю, производная тоже устремится к нулю. Это значит, что если мы обнулим значение логистического сигмоида, мы попадем в область его насыщения, и все станет плохо. А у tanh или, скажем, SoftSign с этим проблем нет: приближая значение к нулю, мы как раз попадаем в окрестность точки $x=0$, где производная максимальна, и из нее легко потом сдвинуться в любую нужную сторону. Таким образом, логистический сигмоид самой своей формой мешает обучать глубокую сеть, причем дело может быть буквально в конкретной параметризации: даже если просто заменить $\sigma(x)$ на функцию $2\sigma(x)-1$, которая будет иметь область значений от -1 до 1 и максимум производной в нуле, эти проблемы могут исчезнуть.

К сожалению, простой заменой одной функции активации на другую проблема обучения глубоких нейронных сетей не решается. Симметричные функции активации тоже страдают от целого ряда проблем. Так, например, последовательные слои с нелинейностью вида tanh поочередно насыщаются: значения нейронов слой за слоем сходятся к 1 или -1, что, как и в случае с обнулением аргумента функции σ, выводит оптимизацию на градиентное плато и часто приводит к «плохому» локальному минимуму. Однако, с другой стороны, такое насыщение куда менее вероятно и после предобучения посредством ограниченных машин Больцмана, и просто после «правильной» случайной инициализации весов.

Все, победа, идеальный и универсальный метод инициализации найден? Не совсем. Хотя мы действительно получили отличный способ инициализации для весов таких нейронов, как выше, на этом наша история про инициализацию весов не заканчивается. Дело в том, что все вышесказанное относится исключительно к *симметричным* функциям активации! Это значит, что инициализация Ксавье будет прекрасно работать для логистического сигмоида или гиперболического тангенса, но в современных сверточных (и не только) архитектурах повсеместно используются и несимметричные функции активации, особенно часто — ReLU.

Очевидно, что инициализация весов для этой функции активации должна отличаться. В 2015 году Каймин Хе (Kaiming He) с соавторами опубликовали работу [114], в которой, помимо нескольких перспективных модификаций ReLU, предложена и подходящая для этой функции активации схема инициализации.

Поскольку функция активации в этом случае несимметрична, в выражении

$$\operatorname{Var}(w_i x_i) = \mathbb{E}[x_i]^2 \operatorname{Var}(w_i) + \mathbb{E}[w_i]^2 \operatorname{Var}(x_i) + \operatorname{Var}(w_i)\operatorname{Var}(x_i)$$

можно сразу обнулить только второй член. И тогда мы получили:

$$\operatorname{Var}(w_i x_i) = \mathbb{E}[x_i]^2 \operatorname{Var}(w_i) + \operatorname{Var}(w_i)\operatorname{Var}(x_i) = \operatorname{Var}(w_i)\,\mathbb{E}\left[x_i^2\right],$$

то есть

$$\operatorname{Var}\left(y^{(l)}\right) = n_{\text{in}}^{(l)} \operatorname{Var}\left(w^{(l)}\right) \mathbb{E}\left[\left(x^{(l)}\right)^2\right],$$

где l обозначает номер текущего уровня, а $n_{\text{in}}^{(l)}$ — число нейронов на уровне l.

Пусть теперь $x^{(l)} = \max(0, y^{(l-1)})$, а $y^{(l-1)}$ распределен симметрично относительно нуля. Тогда:

$$\mathbb{E}\left[\left(x^{(l)}\right)^2\right] = \frac{1}{2}\operatorname{Var}\left(y^{(l-1)}\right),$$

а значит,

$$\operatorname{Var}\left(y^{(l)}\right) = \frac{n_{\text{in}}^{(l)}}{2} \operatorname{Var}\left(w^{(l)}\right) \operatorname{Var}\left(y^{(l-1)}\right).$$

Теперь это очевидным образом приводит к выводу о том, какой должна быть дисперсия инициализации весов для ReLU. Наш окончательный вывод о дисперсии отличается от инициализации Ксавье только тем, что теперь нет никакого n_{out}: если приравнять фактор изменения дисперсии единице, мы получим:

$$\frac{n_{\text{in}}^{(l)}}{2} \operatorname{Var}\left(w^{(l)}\right) = 1 \quad \text{и} \quad \operatorname{Var}(w_i) = \frac{2}{n_{\text{in}}^{(l)}}.$$

Любопытно, что в [114] для ReLU использовалось не равномерное, а нормальное распределение вокруг нуля с соответствующей дисперсией; такой способ инициализации часто используется на практике в глубоких сетях из ReLU-нейронов:

$$w_i \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}^{(l)}}}\right).$$

Кроме того, в обеих работах свободные члены b предлагается инициализировать нулями: они и сами смогут без проблем обучиться практически при любом алгоритме оптимизации.

В качестве вишенки на торте добавим, что авторы работы [114] заметили нюанс, ускользнувший от Глоро и Бенджи, которые пытались найти баланс между дисперсиями прямого и обратного шагов. Предположим, что для инициализации весов сети мы выбрали формулы инициализации Ксавье, но ограничение взяли только из шага обратного распространения ошибки, то есть

$$n_{\text{in}}^{(l)}\text{Var}\,(w_i) = 1, \quad \text{или} \quad \text{Var}\,(w_i) = \frac{1}{n_{\text{in}}^{(l)}}.$$

Тогда, судя по тому, что мы обсуждали выше, дисперсия при прямом шаге вычисления должна будет или «затухать», или «взрываться» пропорционально отношению $n_{\text{in}}^{(l)}/n_{\text{out}}^{(l)}$.

Но ведь в глубокой сети выходы одного слоя подаются на вход следующему, то есть $n_{\text{out}}^{(l)} = n_{\text{in}}^{(l+1)}$! А это значит, что когда мы возьмем произведение по всем слоям сети, окажется, что все члены посередине сокращаются, и фактор изменения дисперсии становится равным

$$\frac{n_{\text{in}}^{(0)}}{n_{\text{out}}^{(0)}}\frac{n_{\text{in}}^{(1)}}{n_{\text{out}}^{(1)}}\frac{n_{\text{in}}^{(2)}}{n_{\text{out}}^{(2)}}\cdots\frac{n_{\text{in}}^{(L-1)}}{n_{\text{out}}^{(L-1)}}\frac{n_{\text{in}}^{(L)}}{n_{\text{out}}^{(L)}} = \frac{n_{\text{in}}^{(0)}}{n_{\text{in}}^{(1)}}\frac{n_{\text{in}}^{(1)}}{n_{\text{in}}^{(2)}}\frac{n_{\text{in}}^{(2)}}{n_{\text{in}}^{(3)}}\cdots\frac{n_{\text{in}}^{(L-1)}}{n_{\text{in}}^{(L)}}\frac{n_{\text{in}}^{(L)}}{n_{\text{out}}^{(L)}} = \frac{n_{\text{in}}^{(0)}}{n_{\text{out}}^{(L)}},$$

где $n_{\text{out}}^{(L)}$ — число выходов последнего слоя сети, а $n_{\text{in}}^{(0)}$ — число входов первого слоя.

Понятно, что каким бы ни было это соотношение, оно уже не является экспоненциальной функцией от числа слоев сети и не приводит к затуханию или чрезмерному увеличению градиентов. Значит, можно использовать для инициализации только ограничение из обратного распространения, а прямое ограничение игнорировать.

Мы уже видели, что в библиотеке Keras реализована возможность инициализации слоев по методу Ксавье. Таким же образом, при помощи параметров `init="he_uniform"` или `init="he_normal"`, можно задать инициализацию Хе, и при этом, как мы объяснили выше, дисперсия в этом случае будет зависеть исключительно от числа нейронов в данном слое.

Краткое резюме у данного раздела получается очень простое: для симметричных функций активации с нулевым средним (в основном tanh) используйте инициализацию Ксавье, а для ReLU и ему подобных — инициализацию Хе.

В результате такой случайной инициализации получится, что предобучение без учителя не то чтобы ничего не улучшает и совсем бессмысленно, но обычно слишком сложно и трудоемко: нужно ведь фактически обучать совсем другую модель, а то и последовательность моделей, часто хрупких и сложных для обучения. Современные улучшения методов оптимизации глубоких нейронных сетей вполне способны заменить предобучение без учителя. Одно из таких улучшений — хорошая случайная инициализация, а к другому мы переходим в следующем параграфе: нас ждет нормализация по мини-батчам.

4.3. Нормализация по мини-батчам

Все нормально. Падаю!

Из к/ф «В бой идут одни старики»

— Нормально, Григорий! — Отлично, Константин!

М. Жванецкий

В этом разделе мы обсудим очередную идею, перевернувшую мир обучения глубоких нейронных сетей. Звучит странно, но это действительно так: идея *нормализации по мини-батчам* (batch normalization), которую мы здесь излагаем, появилась совсем недавно, и ничего такого уж сложного в ней нет. Но быстро оказалось, что она действительно способна улучшить обучение в очень широком спектре архитектур, и сейчас применяется сплошь и рядом. Наше изложение следует исходной статье Сергея Иоффе (Sergey Ioffe) и Кристиана Сегеди (Christian Szegedy) о нормализации по мини-батчам, опубликованной в 2015 году [252]; как видите, в обучении глубоких сетей свежие хорошие идеи могут распространяться очень быстро.

Мы уже много раз обсуждали, что при обучении нейронных сетей один шаг градиентного спуска обычно делается не на одной точке входных данных, а на *мини-батче*, то есть на небольшой коллекции данных, которая обычно выбирается из всего обучающего множества случайно. С точки зрения оптимизации у такого подхода есть сразу несколько преимуществ.

Во-первых, усреднение градиента по нескольким примерам представляет собой апроксимацию градиента по всему тренировочному множеству, и чем больше примеров используется в одном мини-батче, тем точнее это приближение. Максимальная точность апроксимации достигалась бы, безусловно, в том случае, если бы мы каждый шаг делали сразу на всем тренировочном датасете, но такая точность обычно вычислительно недостижима. Во-вторых, глубокие нейронные сети подразумевают большое количество последовательных действий с каждым примером, а современное многопоточное «железо» позволяет эту длинную последовательность действий применять одновременно к достаточно большому числу примеров в параллельном режиме; такой эффект, конечно, особенно важен при перемещении алгоритмов обучения на видеокарты.

Но при этом глубина сетей приводит к следующей проблеме. Если на очередном шаге градиентного спуска меняются веса одного из первых слоев, то это неминуемо приводит к изменению распределения активаций выходов этого слоя. А значит, всем последующим слоям надо адаптироваться к по-новому распределенным данным. Мы проиллюстрировали этот эффект на рис. 4.3, где изображен простейший нейрон первого слоя с двумя входами и нелинейностью в виде tanh:

$$y = \tanh\left(w_0 + w_1 x_1 + w_2 x_2\right).$$

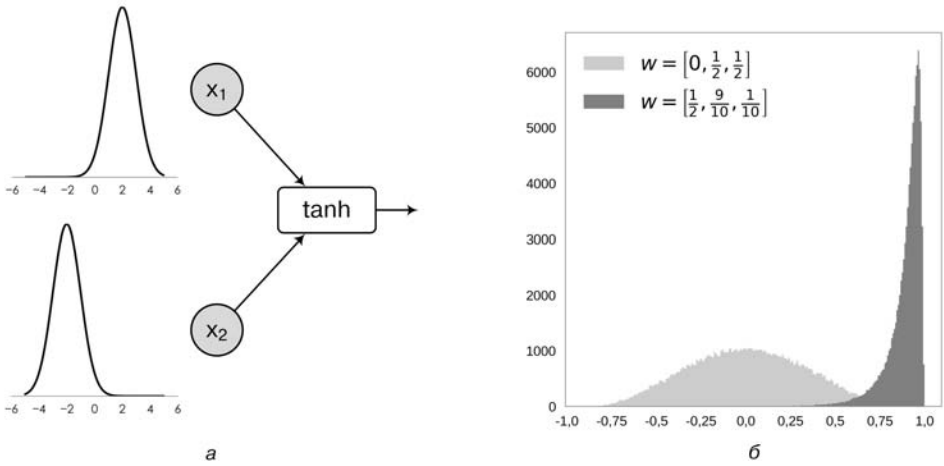


Рис. 4.3. Пример внутреннего сдвига переменных: *а* — структура первого слоя сети и входные распределения; *б* — результат для двух разных векторов весов

На рис. 4.3, *а* показана общая схема вычислений и выбранные нами распределения входов этого нейрона: нормальные распределения со средними 2 и -2 и дисперсией $\frac{1}{2}$. А на рис. 4.3, *б* показано распределение[1] выходов нейрона для двух разных случаев: для весов $\boldsymbol{w} = (w_0, w_1, w_2) = (0, \frac{1}{2}, \frac{1}{2})$ и для весов $\boldsymbol{w} = (\frac{1}{2}, \frac{9}{10}, \frac{1}{10})$. Как видите, результаты совершенно разные, и это абсолютно нормально и ожидаемо: мы действительно хотели бы, чтобы наши нейроны со временем обучались! И логично, что выходы их при обучении изменяются.

Однако что происходит, когда нейрон начинает обучаться, с точки зрения нейронов следующего уровня? Сначала, при векторе весов $(0, \frac{1}{2}, \frac{1}{2})$ и около того, нейрон следующего уровня получал на вход что-то вроде нормального распределения с нулевым средним. Он хорошо обучился реагировать на входы из интервала $[-\frac{1}{2}, \frac{1}{2}]$, а на другие не обучился: и необходимости не было, и возможности, потому что не было таких тренировочных примеров. А потом, когда вектор весов сместился и превратился в $(\frac{1}{2}, \frac{9}{10}, \frac{1}{10})$, выход нейрона первого уровня стал почти всегда попадать в интервал $[\frac{1}{2}, 1]$, причем ближе к единице. Получается, что все то, чему обучался нейрон второго уровня до этого, стало почти бесполезным: его входы теперь берутся из совершенно новой области, и обучаться ему надо фактически заново.

Эта проблема получила название *внутреннего сдвига переменных* (internal covariance shift). Кроме того что нейронам приходится обучаться заново, внутренний сдвиг приводит и к другой проблеме, которую мы уже обсуждали в разделе 3.5. Мы

[1] Распределение здесь эмпирическое: мы посэмплировали десять тысяч точек по входным распределениям и нарисовали гистограмму результатов.

видели, что очень часто в нейронных сетях используются сигмоидальные функции активации, одной из особенностей которых является «насыщение» значений функций активации, когда их входы получают большие по модулю значения. Например, в нашем примере для $f(x) = \tanh(x)$ при увеличении $|x|$ производная $f'(x)$ достаточно быстро стремится к нулю, и выходы нейрона, соответствующие весам $\boldsymbol{w} = (\frac{1}{2}, \frac{9}{10}, \frac{1}{10})$, уже достаточно очевидным образом «приклеиваются» к единице и имеют куда меньшую дисперсию, чем при $\boldsymbol{w} = (0, \frac{1}{2}, \frac{1}{2})$. Изменение распределения значений активации очередного слоя в сторону увеличения абсолютных значений может в таком случае привести к тому, что в последующих слоях функции активации «насытятся», и сквозь их нулевые производные градиентам трудно будет пройти. Отчасти эту проблему можно решить, заменив сигмоидальные функции активации на такие, как ReLU, но это не единственный и не всегда подходящий способ.

Конечно, проблема сдвига переменных не является специфической сугубо для глубоких нейронных сетей. В классическом машинном обучении проблема обычно возникала в такой форме: часто распределение данных в тестовой выборке (то есть уже при применении модели) может существенно отличаться от распределения данных в обучающей выборке. Эта задача была объектом пристального внимания исследователей; см., например, статьи [43, 98, 122, 493, 514] и даже целую отдельную книгу [103], целиком посвященную сдвигу переменных.

Методы предлагались разные, но одним из основных всегда была *нормализация* данных в той или иной форме. В классических нейронных сетях нормализация тоже использовалась: благодаря исследованиям [135, 570] известно, что процесс обучения сходится быстрее, когда входы сети «отбелены» (whitened), то есть их среднее приведено к нулю, а матрица ковариаций — к единичной. В этом состоит и идея нормализации в глубоких сетях: если операцию «отбеливания» применять к входам каждого слоя, это позволит избежать проблемы сдвига переменных.

Нормализация входов часто помогает, и прежде чем обучать нейронные сети, ее тоже желательно сделать. Однако если мы будем на каждом шаге обучения просто нормировать входы очередного слоя внутри сети, без учета этой операции при обучении, ни к чему хорошему это не приведет. Представим себе слой, добавляющий смещение b к своим входам $\boldsymbol{u}$. Мы можем нормализовать его, вычтя среднее значение активации:

$$\hat{\boldsymbol{x}} = \boldsymbol{x} - \mathbb{E}[\boldsymbol{x}],$$

где $\boldsymbol{x} = \boldsymbol{u} + b$, а $\mathbb{E}[\boldsymbol{x}] = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i$ для набора данных из N точек[1].

На очередном шаге градиентного спуска параметр b обновится, $b := b + \Delta b$. Однако легко видеть, что значение $\hat{\boldsymbol{x}}$ останется прежним:

$$\boldsymbol{u} + b + \Delta b - \mathbb{E}[\boldsymbol{u} + b + \Delta b] = \boldsymbol{u} + b - \mathbb{E}[\boldsymbol{u} + b].$$

[1] Кстати, эта конструкция очень похожа на основную идею остаточного обучения, о котором мы поговорим в разделе 5.4.

А это значит, что, несмотря на изменение параметра, нормализованная активация слоя не изменилась! Так что теперь эта ситуация станет повторяться, и в результате абсолютное значение b будет расти совершенно неограниченно.

Данный пример говорит о том, что нормализацию необходимо учитывать при градиентном спуске. Но что это значит с точки зрения распространения градиентов по графу? Давайте введем слой нормализации:

$$\hat{\boldsymbol{x}} = \text{Norm}(\boldsymbol{x}, \mathcal{X}),$$

параметрами которого являются не только текущий обучающий пример $\boldsymbol{x}$, но и все примеры из тренировочной выборки $\mathcal{X}$. Следовательно, для шага градиентного спуска нам необходимо вычислить якобианы $\frac{\partial \text{Norm}}{\partial \boldsymbol{x}}$ и $\frac{\partial \text{Norm}}{\partial \mathcal{X}}$, причем просто пропустить второй из них не получится — мы получим «взрыв» коэффициентов по описанной выше схеме. Кроме того, для самой операции «отбеливания» (декорреляции) нам потребуется вычислить матрицу ковариаций:

$$\text{Cov}[\boldsymbol{x}] = \mathbb{E}_{\boldsymbol{x} \in \mathcal{X}} \left[\boldsymbol{x}\boldsymbol{x}^\top \right] - \mathbb{E}\left[\boldsymbol{x}\right] \mathbb{E}\left[\boldsymbol{x}\right]^\top,$$

затем обратить ее и вычислить из нее квадратный корень, а при градиентном спуске еще и производные такого преобразования. В конечном счете это выливается в непомерные дополнительные вычислительные расходы, и становится очевидным, что нормализовать веса надо как-то по-другому.

Поскольку полноценную декорреляцию для каждого слоя сделать за разумное время невозможно, особенно для больших датасетов, на практике, вместо того чтобы декоррелировать все нейроны слоя совместно, нормализуют каждый элемент входного вектора по отдельности. Это значит, что вектор $\boldsymbol{x}$ после нормализации будет выглядеть так:

$$\hat{\boldsymbol{x}} = (\hat{x}_1, \ldots, \hat{x}_d) \quad \text{где} \quad \hat{x}_k = \frac{x_k - \mathbb{E}\left[x_k\right]}{\sqrt{\text{Var}\left(x_k\right)}}.$$

Среднее и дисперсию в последней формуле хотелось бы, конечно, вычислять сразу по всему датасету, но это будет совершенно невозможно вычислительно — для каждого вычисления потребуется весь датасет! Так что $\mathbb{E}\left[x_k\right]$ и $\text{Var}\left(x_k\right)$ мы будем считать по текущему мини-батчу, и данный подход называется *нормализацией по мини-батчам*. Несмотря на то, что между признаками внутри слоя при таком подходе корреляции могут сохраняться, еще с конца 90-х годов прошлого века известно, что такой подход ускоряет сходимость обучения нейронных сетей [135].

Однако, кроме очевидных достоинств, у такого подхода есть и некоторые не столь очевидные недостатки. Если в качестве функции активации используется сигмоидальная функция, например логистический сигмоид $\sigma(x) = \frac{1}{1+e^{-x}}$, то, когда мы нормализуем ее аргумент, мы увидим, что нелинейность по сути пропадет: подавляющее большинство нормализованных значений будут попадать в область,

где сигмоид ведет себя очень похоже на линейную функцию, и функция активации фактически станет линейной. Мы еще вернемся в последующих главах к тому, что способность нейросети научиться воспроизводить тождественную функцию может играть существенную роль в обучении.

Для того чтобы компенсировать эти недостатки, слой нормализации должен быть способен работать как тождественная функция, то есть при некоторых комбинациях параметров он должен иметь возможность со входами буквально ничего не делать. Чтобы так могло получиться, мы вводим в слой нормализации дополнительные параметры γ_k и β_k для масштабирования и сдвига нормализованной активации по каждой компоненте:

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - \mathbb{E}\left[x_k\right]}{\sqrt{\mathrm{Var}[x_k]}} + \beta_k.$$

Эти параметры будут обучаться вместе со всеми прочими параметрами нейронной сети; они добавляют «степеней свободы», которые позволяют восстановить выразительную способность сети в целом. В частности, теперь слой нормализации может обучиться реализовывать тождественную функцию: теперь для этого достаточно положить $\gamma_k = \sqrt{\mathrm{Var}[x^{(k)}]}$ и $\beta^{(k)} = \mathbb{E}\left[x^{(k)}\right]$.

Последнее небольшое замечание состоит в том, что на практике, чтобы избежать деления на ноль при нормализации, к дисперсии добавляется небольшое постоянное значение ϵ. Теперь мы можем формально описать слой батч-нормализации. Слой получает на вход очередной мини-батч $B = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_m\}$, а затем последовательно:

- вычисляет базовые статистики по мини-батчу:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} \boldsymbol{x}_i, \quad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (\boldsymbol{x}_i - \mu_B)^2;$$

- нормализует входы:

$$\hat{\boldsymbol{x}}_i = \frac{\boldsymbol{x}_i - \mu_b}{\sqrt{\sigma_B^2 + \epsilon}};$$

- вычисляет результат:

$$\boldsymbol{y}_i = \gamma \boldsymbol{x}_i + \beta.$$

Мы могли бы привести формулы для обучения, подсчитав, как градиент функции ошибки будет пропускаться через этот слой и каковы будут производные по новым параметрам γ и β... но не будем: к чему загромождать книгу достаточно прямолинейными вычислениями, которые уже много раз реализованы в любой стандартной библиотеке автоматического дифференцирования. Мы уверены, что к этому моменту заинтересованный читатель вполне способен взять производные от этих функций самостоятельно.

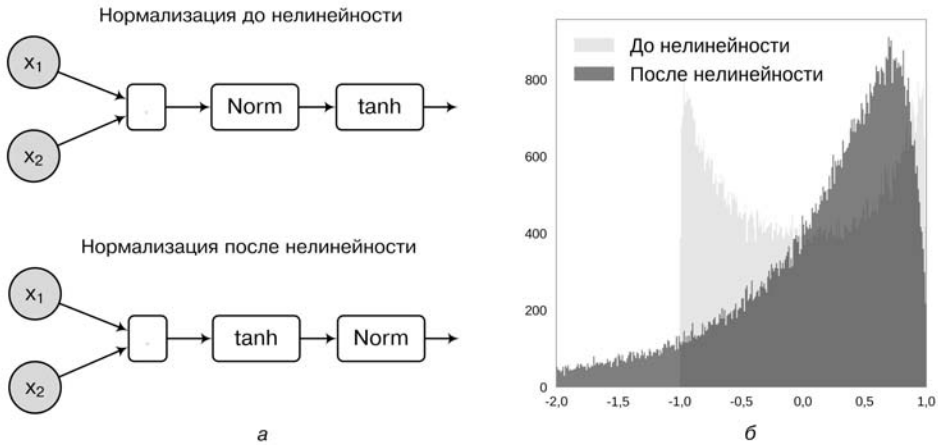


Рис. 4.4. Нормализация до и после нелинейности: *а* — графы вычислений в обоих случаях; *б* — результаты сэмплирования

Пора переходить к практике. Но сначала заметим, что нормализацию можно делать в разные моменты времени. В частности, до сих пор нет устоявшегося мнения по вопросу о том, лучше делать ее после очередного слоя или после линейной части слоя, до нелинейной функции активации.

В качестве иллюстрации к этому на рис. 4.4 изображены результаты нормализации до и после нелинейности в нашем примере с рис. 4.3. Как видите, эффект получается разный; в частности, естественно, область значений в варианте нормализации после сигмоидальной нелинейности будет шире, чем если нормализовать до нелинейности, ведь tanh или σ снова вернут нормализованные результаты на отрезок $[-1,1]$ или $[0,1]$ соответственно.

И теперь у нас все готово для того, чтобы перейти к эксперименту и наглядно показать на примере, зачем нормализация по мини-батчам вообще нужна. Мы будем проверять ее эффективность на нашем обычном примере, классификации рукописных цифр на датасете MNIST.

Для этого мы построим простую полносвязную модель с дополнительным слоем нормализации по мини-батчам. В этом разделе пример будет на чистом TensorFlow, но в Keras, естественно, нормализация по мини-батчам тоже реализована — там это был бы просто дополнительный слой под названием `keras.layers.normalization.BatchNormalization`.

Сначала импортируем все, что нужно:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Чтобы не выполнять много раз однотипные операции, зададим функции для объявления полносвязного слоя и слоя нормализации по мини-батчам. Полносвязный слой задается уже давно знакомым нам образом:

```
def fullyconnected_layer(tensor, input_size, out_size):
    W = tf.Variable(tf.truncated_normal([input_size, out_size], stddev=0.1))
    b = tf.Variable(tf.truncated_normal([out_size], stddev=0.1))
    return tf.nn.tanh(tf.matmul(tensor, W) + b)
```

А для того, чтобы задать слой нормализации, сначала нужно вычислить соответствующие статистики и объявить переменные, а дальше, как это часто бывает, использовать готовую функцию из библиотеки TensorFlow:

```
def batchnorm_layer(tensor, size):
    batch_mean, batch_var = tf.nn.moments(tensor, [0])
    beta = tf.Variable(tf.zeros([size]))
    scale = tf.Variable(tf.ones([size]))
    return tf.nn.batch_normalization(
        tensor, batch_mean, batch_var, beta, scale, 0.001)
```

Для эксперимента нам будет достаточно очень простой модели: мы возьмем полносвязную нейронную сеть с размерами слоев 784, 100, 100 и 10. Первый слой — входы, куда мы подаем значения пикселов изображения, последний — выходной слой, по одному нейрону на каждый из возможных классов изображения (то есть рукописные цифры от 0 до 9). Между промежуточными слоями мы и вставим слой нормализации по мини-батчам.

```
x = tf.placeholder(tf.float32, [None, 784])
h1 = fullyconnected_layer(x, 784, 100)
h1_bn = batchnorm_layer(h1, 100)
h2 = fullyconnected_layer(h1_bn, 100, 100)
y_logit = fullyconnected_layer(h2, 100, 10)
```

Для обучения осталось только задать функцию ошибки и метод оптимизации:

```
loss = tf.nn.sigmoid_cross_entropy_with_logits(y_logit, y)
train_op = tf.train.GradientDescentOptimizer(0.01).minimize(loss)
```

Чтобы оценить эффект от добавления слоя нормализации по мини-батчам, мы сравнили описанную выше модель с идентичной, но без дополнительного слоя нормализации. Графики на рис. 4.5 устроены так же, как на рис. 4.2: по горизонтальной оси отложено число эпох обучения, а по вертикальной — значение функции ошибки (слева) и точность (accuracy) предсказания модели на тестовом множестве (справа). Снова затененная область вокруг кривых показывает дисперсию. Видно, что модель со слоем нормализации по мини-батчам практически сразу показывает хорошие результаты и быстро приближается к отметке в 98 % точности, в то время как обычная модель добивается прогресса гораздо медленнее.

Только появившись, нормализация по мини-батчам наделала немало шума. Для большинства классических архитектур добавление нормализации дало очень

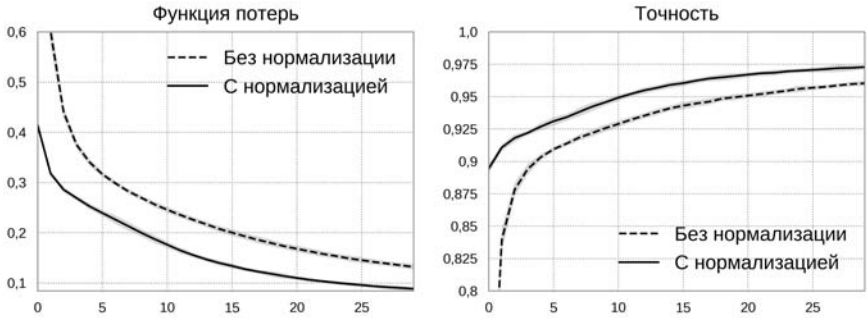


Рис. 4.5. Сравнение сети с нормализацией по мини-батчам и без нее

мощный толчок вперед. Именно нормализация по мини-батчам позволила обучать такие очень глубокие архитектуры, как Inception и ResNet. Буквально за год нормализация по мини-батчам не только вошла в базовый инструментарий специалистов по глубокому обучению, но и стала одним из ключевых элементов при обучении по-настоящему глубоких нейронных сетей. В оставшейся части этого раздела мы рассмотрим несколько самых последних результатов о нормализации, которые продолжают и развивают идею нормализации по мини-батчам.

Первой проблемой с нормализацией по мини-батчам стало то, что с рекуррентными сетями, в отличие от полносвязных и сверточных, все оказалось не так просто[1]. Оставим за скобками различные архитектуры рекуррентных модулей и рассмотрим только наиболее общую конструкцию: получая на вход последовательность векторов $(\boldsymbol{x}_1,...,\boldsymbol{x}_T)$, рекуррентная сеть выдает последовательность скрытых состояний $(\boldsymbol{h}_1,...,\boldsymbol{h}_t)$, вычисляя их шаг за шагом в таком виде:

$$\boldsymbol{h}_t = f\left(W_h \boldsymbol{h}_{t-1} + W_x \boldsymbol{x}_t\right),$$

где f — нелинейная функция активации, W_h — матрица рекуррентных весов, а W_x — матрица весов для входов.

Подробные исследования показали [34], что использование слоя нормализации по мини-батчам «в лоб» не приносит пользы рекуррентным сетям. Эксперименты с нормализацией в следующей форме:

$$\boldsymbol{h}_t = f\left(\mathrm{BN}(W_h \boldsymbol{h}_{t-1}) + W_x \boldsymbol{x}_t\right)$$

показали результаты несколько хуже, чем у чистой рекуррентной сети.

[1] Очень логично обсудить современные результаты и варианты нормализации именно здесь, но чтобы их объяснить, нам придется ссылаться на общее понимание того, что такое рекуррентные нейронные сети. Подробно об этом мы поговорим в главе 6, а сейчас, если остаток раздела вызывает сложности, его можно пропустить без проблем для дальнейшего понимания.

Одна из основных причин такого провала — то, что распределение активаций скрытого слоя может сильно меняться при последовательном применении одного и того же рекуррентного преобразования к последовательности входов. По сути это значит, что есть смысл выравнивать скрытое состояние сети «сквозь время», так как именно разница между состояниями на разных шагах обработки последовательности и является основной идеей и смыслом работы рекуррентной сети.

Поэтому первый положительный результат применения нормализации по мини-батчам к рекуррентным сетям был получен командой авторов из университета Монреаля весной 2016 года [439]. Ключевой идеей здесь стало применение нормализации для каждого момента времени по отдельности. По сути, мы обучаем не пару параметров β и γ, а наборы $\boldsymbol{\beta} = (\beta_1,...,\beta_T)$ и $\boldsymbol{\gamma} = (\gamma_1,...,\gamma_T)$. Кроме того, авторы отделили нормализацию скрытого слоя от нормализации входов, предложив вариант:

$$\boldsymbol{h}_t = f(\text{BN}(W_h\boldsymbol{h}_{t-1};\beta_h,\gamma_h) + \text{BN}(W_x x_t;\beta_x,\gamma_x)),$$

что в итоге привело к существенному ускорению обучения и улучшению качества моделей на такой задаче, как побуквенное моделирование языка (мы ее подробно обсудим в разделе 6.6).

Несмотря на огромный успех нормализации по мини-батчам, ряд проблем у этого подхода все же оставался. Трудности использования слоя нормализации в рекуурентных сетях не ограничиваются тем, что нужно обучать отдельные параметры для каждого временного шага. В частности, естественной для рекуррентных сетей является переменная длина входной последовательности, когда T различается от входа к входу (например, на вход могут подаваться предложения на естественном языке, закодированные пословно, то есть T — число слов в предложении). Это значит, что нормализацию более поздних временных шагов может быть не так просто обучить: большинство примеров в тренировочном множестве будут небольшой длины, что приводит к серьезным ограничениям на применение обученной рекуррентной нейронной сети, ведь на более поздних элементах последовательности мы уже не сможем применять нормализацию.

Кроме того, если рекурентный модуль независимо от длины входной последовательности имеет фиксированное число обучаемых параметров, то нормализация по мини-батчам с привязкой ко времени добавляет число параметров, пропорциональное максимальной длине входной последовательности — а это значит, что параметров нормализации может для длинных последовательностей стать запредельно много. И даже если оставить в стороне проблемы применения нормализации по мини-батчам именно к рекуррентным сетям, все еще остается ограничение, накладываемое слоем нормализации на размер батча, ведь для хорошей оценки статистик может потребоваться большое число примеров. В частности, очевидно, что нормализацию по мини-батчам невозможно применять, когда обучение производится на каждом примере в отдельности.

Все эти проблемы привели к новому взгляду на нормализацию, предложенному летом 2016 года исследователями из университета Торонто (как обычно, без Джеффри Хинтона не обошлось) [16]. В этой работе предлагается нормализовать активации сети не «по мини-батчу», а «по слою», используя отдельные статистики для каждого элемента последовательности, но обучая общие параметры:

$$\boldsymbol{h}^t = f\left(\frac{g}{\sigma^t}\left(\boldsymbol{a}^t - \mu^t + b\right)\right),$$

где $\boldsymbol{a}^t = W_h\boldsymbol{h}^{t-1} + W_x\boldsymbol{x}^t$ — активации до применения нелинейности f, а статистики μ^t и σ^t вычисляются по активациям слоя как

$$\mu^t = \frac{1}{d}\sum_{i=1}^{H} a_i^t, \quad \sigma^t = \sqrt{\frac{1}{d}\sum_{i=1}^{H}(a_i^t - \mu^t)^2},$$

где H — размер скрытого слоя. Параметры g и b обучаются в ходе обучения сети совместно с основными параметрами. На ряде задач машинного обучения с применением рекуррентных сетей такой подход к нормализации показал результаты, сравнимые с батч-нормализацией, или несколько лучше, однако при этом нормализация по слою позволила сократить время обучения и избежать проблем при работе с маленькими размерами батчей.

Как видите, нормализация помогает в разных формах. Мы уже обсудили нормализацию активации нейронов по примерам в мини-батче и по нейронам в слое. Остался последний важный элемент — веса сети.

Недавно Тим Салиманс и Дьедерик Кингма из OpenAI опубликовали исследование влияния нормализации весов на процесс обучения нейронной сети. Предложенный ими подход аналогичен нормализации по батчу или по слою, однако коэффициент репараметризации (нормализации) вычисляется с учетом весов очередного слоя, а не его активации:

$$\boldsymbol{h}_i = f\left(\frac{\gamma}{\|\boldsymbol{w}_i\|}\boldsymbol{w}_i^\top\boldsymbol{x} + b_i\right),$$

где $\boldsymbol{w}_i$ — веса линейной комбинации i-го нейрона, b_i — его смещение, а $\|\boldsymbol{w}_i\|$ обозначает евклидову норму вектора весов. После такой репараметризации норма вектора весов оказывается в точности равна γ, и этот параметр нейронная сеть тоже обучает вместе с основными.

Обозначим «новые» веса сети через $\boldsymbol{v} = \frac{\gamma}{\|\boldsymbol{w}\|}\boldsymbol{w}$. Тогда градиент функции ошибки L по γ равен

$$\nabla_\gamma L = \frac{\boldsymbol{w}^\top\nabla_{\boldsymbol{v}}L}{\|\boldsymbol{w}\|},$$

где $\nabla_{\boldsymbol{v}}L$ представляет собой обычный градиент по весам сети.

А градиент функции ошибки по исходным весам равен

$$\nabla_{\boldsymbol{w}} L = \frac{\gamma}{\|\boldsymbol{w}\|}\nabla_{\boldsymbol{v}} L - \frac{\gamma\nabla_{\gamma} L}{\|\boldsymbol{w}\|^2}\boldsymbol{w},$$

и это, подставив выражение для $\nabla_{\gamma} L$, можно переписать в виде

$$\nabla_{\boldsymbol{w}} L = \frac{\gamma}{\|\boldsymbol{w}\|}M_{\boldsymbol{v}}\nabla_{\boldsymbol{v}} L,$$

где $M_{\boldsymbol{v}} = \mathbb{I} - \frac{\boldsymbol{v}\boldsymbol{v}^{\top}}{\|\boldsymbol{v}\|^2}$ обозначает матрицу проекции на дополнение вектора $\boldsymbol{v}$. По сути, нормализация весов делает две вещи:

- масштабирует градиент с весом $g/\|\boldsymbol{w}\|$;
- «отворачивает» градиент от вектора текущих весов.

Оба эти эффекта приближают матрицу ковариаций градиентов к единичной, что позволяет добиться определенных преимуществ при обучении.

Давайте подробнее разберем, что происходит с весами при таком подходе. Пусть на очередном шаге обучения мы обновляем веса по естественной формуле $\boldsymbol{w}' = \boldsymbol{w} + \Delta\boldsymbol{w}$, где $\Delta w \propto \nabla_w L$. Тогда из-за ортогональности градиентов весам с каждым обновлением норма $\|\boldsymbol{w}\|$ будет, согласно теореме Пифагора, постепенно расти. Точнее говоря, если $\frac{\|\Delta\boldsymbol{w}\|}{\|\boldsymbol{w}\|} = c$, то $\|\boldsymbol{w}'\| = \sqrt{1+c^2}\|\boldsymbol{w}\|$. Скорость роста при этом будет зависеть от дисперсии градиентов: чем больше дисперсия, тем больше $\|\Delta\boldsymbol{w}\|$, а следовательно, и c, что в свою очередь приводит к уменьшению $\frac{\gamma}{\|\boldsymbol{w}\|}$. Если же дисперсия градиентов невелика, то и $\sqrt{1+c^2} \approx 1$.

В итоге этот механизм позволяет нейронной сети самостоятельно стабилизировать норму градиентов. Кроме того, приятным дополнением оказался эмпирически установленный факт устойчивости к выбору скорости обучения. Фактически получилось, что благодаря возможности менять норму весов и тем самым коэффициент масштабирования градиентов сеть оказалась способна компенсировать слишком большие или слишком маленькие значения параметра скорости обучения.

А в работе [482] несколько немецких ученых из группы Юргена Шмидхубера разработали так называемые *самонормализующиеся* нейронные сети (self-normalizing neural networks, SNNs). Оказалось, что можно добавить свойства нормализации непосредственно в функцию активации нейрона, и для этого достаточно просто использовать масштабированную экспоненциальную функцию:

$$\mathrm{SELU}(x) = \lambda\begin{cases} x, & \text{если } x > 0, \\ \alpha e^x - \alpha, & \text{если } x \le 0. \end{cases}$$

Это совсем свежая работа, и пока непонятно, вырастет ли из этого еще одна мини-революция в нейронных сетях, но результаты очень интересные.

Подведем краткий итог. Несмотря на то что первая публикация о нормализации активаций появилась только в конце 2015 года, уже сейчас ни одна современная глубокая архитектура не обходится без того или иного способа нормализации. Все обучение в глубоких нейронных сетях ведется так или иначе при помощи градиентного спуска. Поэтому появление новых идей в основаниях этой области, то есть изменений, улучшающих сам процесс оптимизации, может привести к ошеломляющим результатам. Так получилось с нормализацией по мини-батчам: по сути, ее применение позволило перейти от архитектур глубиной не более пары десятков слоев к сверхглубоким архитектурам, которые сейчас насчитывают сотни и даже тысячи слоев.

Но это был не единственный прорыв в основаниях оптимизации в нейронных сетях. Важнейшую роль для современного глубокого обучения играет и сам процесс градиентного спуска: его современные модификации работают во много раз лучше классических подходов, и сейчас мы как раз о них и поговорим.

4.4. Метод моментов: Ньютон, Нестеров и Гессе

> Поднявшись на высоту 2000 м, он нагнал австрийца сверху и задел его своим шасси... Аппарат Нестерова после этого стал спирально спускаться.
>
> *В. Рохмистров. Авиация великой войны*

Итак, мы разобрались с тем, как делать градиентный спуск, и с тем, как осуществлять это на практике: стохастический градиентный спуск, в том числе вариант с мини-батчами, действительно работают на практике и позволяют обучать нейронные сети. Однако вспомним, что у градиентного спуска был свободный параметр: скорость обучения η показывала, насколько сильно мы сдвигаем параметры в сторону очередного градиента.

Скорость обучения — это чрезвычайно важный параметр. Если она будет слишком большой, то алгоритм станет просто прыгать по фактически случайным точкам пространства и никогда не попадет в минимум, потому что все время будет через него перепрыгивать. А если она будет слишком маленькой, то такой проблемы не будет, но, во-первых, обучение станет гораздо медленнее, а во-вторых, алгоритм рискует успокоиться и сойтись в первом же локальном минимуме, который вряд ли окажется самым лучшим.

Кажется интуитивно очевидным, что скорость обучения должна сначала быть большой, чтобы как можно быстрее прийти в правильную область пространства поиска, а потом стать маленькой, чтобы уже более детально исследовать окрестности точки минимума и в конце концов попасть в нее. Поэтому простейшая стратегия управления скоростью обучения так и выглядит: мы начинаем с большой

скорости η_0 и постепенно уменьшаем ее по мере того, как продвигается обучение. Часто для этого используют или линейное затухание:

$$\eta = \eta_0 \left(1 - \frac{t}{T}\right),$$

или экспоненциальное:

$$\eta = \eta_0 e^{-\frac{t}{T}},$$

где t — это прошедшее с начала обучения время (число мини-батчей или число эпох обучения), а T — параметр, определяющий, как быстро будет уменьшаться η.

Если правильно подобрать параметры η_0 и T, такая стратегия будет почти наверняка работать лучше, чем градиентный спуск с постоянной скоростью, а если повезет, то и вообще работать будет хорошо.

Однако, во-первых, мы заменили подбор одного параметра η двумя, η_0 и T, то есть не то чтобы сильно упростили себе задачу. Во-вторых, такое постепенное замедление обучения с фиксированными параметрами совершенно никак не отражает собственно форму и характер функции, которую мы оптимизируем. Можно ли улучшить стохастический градиентный спуск, решив эти проблемы?

Оказывается, что учесть форму функции можно, и так называемые *адаптивные* методы градиентного спуска, которые меняют параметры спуска в зависимости от происходящего с функцией, будут работать еще лучше. Идея здесь заключается в том, чтобы вместо глобальной скорости обучения, которая меняется по общей формуле и никак не связана с собственно оптимизируемой функцией, попробовать учитывать в скорости обучения непосредственно «ландшафт» функции, данный нам в ощущениях градиентов в различных точках.

Так, например, в местах, где поверхность функции больше наклонена в одном измерении, чем в другом, обычный стохастический градиентный спуск может вести себя некорректно. Например, если минимум находится в сильно вытянутом «овраге», шаг градиентного спуска будет направлен от одной близкой и крутой стенки этого оврага к другой. А когда точка попадет на другую стенку, градиент станет направлен в противоположную сторону. Получается, что в процессе обучения мы будем все время прыгать туда-сюда с одной стенки оврага на другую, но к общему минимуму будем продвигаться очень медленно; эту ситуацию мы уже обсуждали в разделе 2.4 и даже иллюстрировали на рис. 2.6. И это не надуманный «худший случай»: такое часто случается поблизости от локальных минимумов (и максимумов), поэтому от подобной проблемы хотелось бы избавиться.

Метод импульсов помогает ускорить градиентный спуск в нужном направлении и уменьшает его колебания. Представьте себе, что точка не просто подчиняется правилам градиентного спуска, а действительно движется по ландшафту оптимизируемой функции. Тогда, если мы на секунду представим, что в многомерном пространстве поиска действуют те же законы ньютоновской механики, что и в нашем бренном мире, обладающая массой точка (соответствующая текущему

значению вектора весов при обучении) действительно будет стремиться двигаться согласно законам градиентного спуска: ее ускорение будет направлено в сторону, обратную градиенту функции, описывающей поверхность. Но как только точка начнет собственно движение, классический градиентный спуск перестанет быть применим: если точка пришла в очередное положение с какой-то уже ненулевой скоростью, то эта скорость не сможет мгновенно измениться на противоположную, ускорение будет направлено по градиенту, но само движение будет поначалу продолжаться в ту же сторону, в которую и было направлено. Проще говоря, материальные точки обладают инерцией, а их момент движения нельзя изменить мгновенно.

Этот эффект инерции и пытается выразить метод моментов. Вместо того чтобы двигаться строго в направлении градиента в конкретной точке, мы стараемся продолжить движение в том же направлении, в котором двигались раньше; но градиент, конечно, тоже участвует в формировании окончательной «скорости движения» вектора аргументов. Отсюда и название метода: у нашей «материальной точки», которая спускается по поверхности, появляется импульс, она движется по инерции и стремится этот импульс сохранить. А формально уравнение для вектора разности параметров, который мы используем для обновления весов, теперь выглядит вот так:

$$u_t = \gamma u_{t-1} + \eta \nabla_\theta E(\theta),$$
$$\theta = \theta - u_t.$$

Здесь γ — это параметр метода моментов; он всегда меньше единицы, и он определяет, какую часть прошлого градиента мы хотим взять на текущем шаге, а на какую часть будем использовать новый градиент. Теперь, когда наш «шарик» катится с горки, он все больше ускоряется в том направлении, в котором были направлены сразу несколько предыдущих градиентов, но будет двигаться достаточно медленно в тех направлениях, где градиент все время меняется. В псевдокоде это выглядит примерно так:

```
u = gamma*u - learning_rate * grad
theta += u
```

Однако, продолжая аналогию, хотелось бы не просто слепо бежать вниз в подходящем направлении, а еще и хотя бы на полшага вперед смотреть под ноги, чтобы не споткнуться о внезапно подвернувшийся камень. Поэтому *метод Нестерова*[1]

[1] *Нестеров, Юрий Евгеньевич* (р. 1956) — советский, а теперь российский математик, один из самых известных специалистов в теории оптимизации. Нестеров — создатель ряда очень популярных алгоритмов оптимизации, автор классического учебника по выпуклой оптимизации [386], лауреат премий Данцига и фон Неймана. Сейчас Нестеров работает в Лувенском католическом университете (Universit catholique de Louvain), одном из важнейших бельгийских университетов, впервые основанном в 1425 году (потом, правда, Лувенский университет надолго закрывали).

использует при расчете градиента значение функции ошибки не в точке θ, как выше, а в точке $\theta - \gamma u_{t-1}$. Это справедливо, если учесть, что γu_{t-1} согласно методу моментов уже точно будет использовано на этом шаге, а значит, и изменившийся градент разумно считать уже в той точке, куда мы придем после применения момента предыдущего шага.

Математически вектор обновления параметров теперь можно записать так:

$$u_t = \gamma u_{t-1} + \eta \nabla_\theta E(\theta - \gamma u_{t-1}).$$

Но и это еще не все. Еще более «продвинутый» подход к вычислению моментов — это *метод Ньютона*. Это метод второго порядка, то есть он использует информацию о вторых производных функции, а не только первых. Смысл метода Ньютона прост: представьте, что вы хотите оптимизировать некоторую сложную функцию. Градиент — это линейное приближение функции в данной точке; мы выясняем, в каком направлении функция быстрее всего меняется в нужную сторону, и сдвигаемся туда. Но можно сделать и следующий шаг: давайте приблизим функцию в текущей точке не линейной функцией, а квадратичной, то есть параболоидом. Сделать это можно по формуле Тейлора — неподалеку от точки $\boldsymbol{\theta}_0$:

$$E(\boldsymbol{\theta}) \approx E(\boldsymbol{\theta}_0) + \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top H(E(\boldsymbol{\theta}))(\boldsymbol{\theta} - \boldsymbol{\theta}_0),$$

где $H(E(\boldsymbol{\theta}))$ — это матрица вторых производных функции E, так называемый *гессиан*, или матрица Гессе[1]. И теперь, когда мы приблизили $E(\boldsymbol{\theta})$ параболоидом, можно не сдвигаться на непонятное расстояние вдоль градиента, а сразу смело переходить в точку минимума параболоида, даже если она расположена далеко; эту точку в любом случае легко подсчитать как решение линейной системы уравнений.

Итак, обновление момента можно записать так:

$$u = -[H(E(\boldsymbol{\theta}))]^{-1} \nabla E(\boldsymbol{\theta}).$$

У методов второго порядка есть два важных преимущества. Во-первых, оптимизация действительно обычно становится гораздо быстрее. Во-вторых, скорость теперь настраивается автоматически, а в уравнении совершенно отсутствует параметр η, что тоже сильно облегчает нам жизнь. Кроме того, казалось бы, найти вторую производную не сильно сложнее, чем первую: нужно просто взять производную еще раз, а это мы умеем делать автоматически по графу вычислений.

К сожалению, на практике оказывается, что вычислять матрицу Гессе все-таки слишком дорого для обучения больших нейронных сетей, ведь даже если предполагать, что производных считаются быстро, нам все равно нужно заполнить целую

[1] *Людвиг Отто Гессе* (Ludwig Otto Hesse, 1811–1874) — немецкий математик, специалист по алгебраическим инвариантам и геометрии. В честь Гессе названа масса объектов в алгебре и геометрии: матрица Гессе, нормальная форма Гессе, группа Гессе, пары Гессе, пучок Гессе...

матрицу, то есть сложность здесь растет квадратично относительно числа параметров. Так называемые *квази-ньютоновские методы*, например классический алгоритм L-BFGS [330], накапливают историю вычисленных градиентов и используют ее для того, чтобы аппроксимировать вторые производные. Однако здесь мы даже не будем подробно объяснять, как они это делают, — к сожалению, для этого им приходится считать производную на всем тренировочном множестве, которое может быть очень большим. Сейчас методы второго порядка и квази-Ньютоновские методы почти не используются в обучении глубоких сетей. Однако найти способ использовать их было бы очень заманчиво, и нам хотелось бы отметить это как одну из важных пока не решенных задач глубокого обучения.

Но пока она остается нерешенной, нам придется повозиться со скоростью обучения η. При обучении нейронных сетей полезно постепенно уменьшать размер шага. Можно представить, что при большом размере параметра η наш «шарик» имеет слишком большую кинетическую энергию, и параметры системы очень сильно «прыгают» и постоянно меняют свои значения. В результате шарик не может найти глубокие, но узкие провалы в поверхности функции потерь. А в обратной ситуации, со слишком маленьким размером шага, обучение будет продвигаться слишком медленно; все это мы видели на рис. 2.6.

Чтобы решить сложившуюся проблему, нужно так или иначе уменьшать размер шага по мере обучения: таким образом мы сначала постараемся найти большую область, «долину», где функция в целом имеет не слишком большие значения, а затем уже будем искать в этой долине более точное значение минимума. Для этого нам нужно решить, как именно и когда уменьшать скорость обучения: плохая стратегия уменьшения скорости обучения не решит проблем слишком маленького и слишком большого η, а может их и усугубить. Самый часто используемый метод — это пошаговое уменьшение: раз в несколько эпох η домножается на какую-либо константу d, выбранную в начале обучения; d, естественно, должна быть меньше единицы. Другой хороший метод — начинать уменьшать размер шага после того, как прекращает уменьшаться ошибка на валидационном множестве.

Конечно, существуют и другие методы. Например, можно делить размер шага на номер итерации (линейно уменьшая скорость) или домножать на экспоненту от номера итерации, умноженного на отрицательную константу (уменьшая скорость экспоненциально), но на практике пошаговое уменьшение оказывается самым легко интерпретируемым и потому популярным способом. Баланс здесь примерно такой: большая скорость обучения в начале и слишком быстрое ее уменьшение приведут к большей дисперсии результатов; получится, что модель быстро «выбирает» долину, в которой ей повезло оказаться, и дальше уже просто немножко дообучает результат. А медленный спуск будет выбирать долину более тщательно и может привести к более хорошим результатам, но делать это он будет гораздо дольше. Так что если у вас есть достаточно много вычислительных ресурсов и нет сильных ограничений по времени, можно изначально взять небольшой размер шага и уменьшать его достаточно плавно — не промахнетесь.

4.5. Адаптивные варианты градиентного спуска

Я должен ступать осторожно.

В. В. Набоков. Лолита

В методах, которые мы обсуждали до сих пор, шаг обновления зависел только от текущего градиента и глобального параметра скорости обучения, но никак не учитывал историю обновлений для каждого отдельного параметра.

Однако вполне может так сложиться, что некоторые веса уже близки к своим локальным минимумам, и по этим координатам нужно двигаться медленно и осторожно, а другие веса еще на середине соответствующего склона, и их можно менять гораздо быстрее. К тому же, само по себе регулирование скорости обучения может оказаться довольно затратным делом. Для того чтобы иметь возможность адаптировать скорость обучения для разных параметров автоматически, были созданы *адаптивные* методы оптимизации. Несмотря на то, что у них все равно есть свои собственные метапараметры, эти методы, как правило, ведут себя лучше на разреженных данных и более стабильны, чем изменение единой скорости обучения в чистом виде.

Первый из адаптивных методов оптимизации, Adagrad [131], основан на следующей идее: шаг изменения должен быть меньше у тех параметров, которые в большей степени варьируются в данных, и, соответственно, больше у тех, которые менее изменчивы в разных примерах. Именно из-за этой особенности Adagrad особенно полезен, когда данные сильно разрежены.

Обозначим через $g_{t,i}$ градиент функции ошибки по параметру θ_i:

$$g_{t,i} = \nabla_{\theta_i} L(\theta).$$

Тогда обновление параметра θ_i на очередном шаге градиентного спуска можно записать так:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i},$$

где G_t — диагональная матрица, каждый элемент которой — это сумма квадратов градиентов соответствующего параметра за предыдущие шаги, то есть:

$$G_{t,ii} = G_{t-1,ii} + g_{t,i}^2,$$

а ϵ — это сглаживающий параметр, позволяющий избежать деления на ноль. Интересно, что если не брать квадратный корень в знаменателе, алгоритм действительно будет работать хуже.

Поскольку операции ко всем координатам применяются одинаковые, мы можем векторизовать эти выражения так:

$$\boldsymbol{u}_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}}\boldsymbol{g}_{t-1}.$$

Здесь и далее мы будем считать, что операции взятия корня и умножения продолжаются на векторы покомпонентно.

В псевдокоде Adagrad может выглядеть примерно так:

```
g_cached += grad**2
u = -learning_rate * grad / (np.sqrt(g_cached) + eps)
theta += u
```

Один из плюсов Adagrad состоит в том, что о параметре скорости обучения η можно больше не волноваться, так как диагональные элементы G по сути и являются индивидуальными скоростями обучения для каждого θ. Поэтому для η обычно используют стандартное значение $\eta = 0{,}01$, но и оно не имеет большого значения, так что настраивать его не обязательно.

А главный минус можно заметить, если обратить внимание на то, что g^2 всегда положительно, а значит, значения G постоянно увеличиваются. Это приводит к тому, что скорость обучения порой уменьшается слишком быстро и в итоге становится слишком маленькой, что плохо сказывается на обучении глубоких нейронных сетей. Еще один минус состоит в том, что глобальную скорость обучения в Adagrad нужно выбирать руками, самостоятельно, и она может оказаться хороша для одних размерностей, но плоха для других.

Adadelta [584] — это довольно несложная, но эффективная модификация алгоритма Adagrad, основная цель которой состоит в том, чтобы попытаться исправить два этих недостатка. Для этого используются две основных идеи. Первая идея довольно проста: чтобы не накапливать сумму квадратов градиентов по всей истории обучения, давайте будем считать их по некоторому окну, а еще лучше по всей истории, но с экспоненциально затухающими весами. Поскольку хранить целую историю предыдущих градиентов нам никто не даст, реализовать это проще всего уже известным способом — введением параметра инерции. Теперь у нас на каждый оптимизируемый параметр найдется свой метапараметр, и если мы обозначим этот метапараметр через ρ, то изменение матрицы G можно будет записать так:

$$G_{t,ii} = \rho G_{t-1,ii} + (1 - \rho)g_{t,i}^2.$$

Как и в методе моментов, ρ должно быть, конечно, меньше 1.

Все остальное происходит в точности так же, как в Adagrad:

$$\boldsymbol{u}_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}}\boldsymbol{g}_{t-1}$$

Теперь G представляет собой экспоненциальное среднее квадратов градиентов, то есть на каждом шаге мы берем в G только часть нашей «истории изменений», а веса старых значений градиентов быстро (экспоненциально быстро!) уменьшаются. Так что экспоненциальное среднее, в отличие от суммы, будет убывать только тогда, когда убывающими станут градиенты. А это именно то, чего мы и добивались — уменьшения скорости обучения в тот момент, когда изменение целевой функции замедляется, для более тонкой настройки вокруг локального минимума.

Второе изменение, внесенное в Adadelta, представляет собой более сложную идею, опять связанную с матрицей Гессе. Дело в том, что этот алгоритм исправляет несовпадающие масштабы, «единицы измерения» параметров и их обновлений в методе градиентного спуска и других его модификациях. Для интуитивного понимания здесь действительно работает эта простая физическая аналогия: давайте представим, что веса измеряются в секундах. Оказывается, что в предыдущих методах «единицы измерения» безнадежно не совпадали:

- в обычном градиентном спуске или методе моментов получалось, что «единицы измерения» обновления параметров $\Delta\theta$ — это единицы измерения градиента, то есть в единицах $\frac{\partial f}{\partial x}$: если веса измерялись в секундах, а целевая функция в метрах, то градиент будет иметь размерность «метр в секунду», и вычитать метры в секунду из секунд довольно странно;
- а в Adagrad получалось, что значения обновлений $\Delta\theta$ зависели от отношений градиентов, то есть величина обновлений получалась и вовсе безразмерной.

Правильные «единицы измерений» получались только в методах второго порядка: в методе Ньютона второго порядка обновление параметров $\Delta\theta$ было пропорционально $H^{-1}\nabla_\theta f$, то есть размерность была такая:

$$\Delta\theta \propto H^{-1}\nabla_\theta f \propto \frac{\frac{\partial f}{\partial \theta}}{\frac{\partial^2 f}{\partial \theta^2}} \propto \text{ размерность } \theta.$$

В вышеописанном примере получались бы метры в секунду, деленные на метры в секунду в квадрате (единицы ускорения), то есть в результате снова получаются секунды.

Чтобы привести масштабы этих величин в соответствие, достаточно домножить обновление из Adagrad на еще один новый сомножитель: еще одно экспоненциальное среднее, но теперь уже от квадратов обновлений параметров, а не от градиента.

Поскольку настоящее среднее квадратов обновлений нам неизвестно, то чтобы его узнать, нам нужно как раз сначала выполнить текущий шаг алгоритма, — оно аппроксимируется предыдущими шагами:

$$\mathbb{E}\left[\Delta\theta^2\right]_t = \rho\mathbb{E}\left[\Delta\theta^2\right]_{t-1} + (1-\rho)\Delta\theta^2, \quad \text{где} \quad u_t = -\frac{\sqrt{\mathbb{E}\left[\Delta\theta^2\right]_{t-1} + \epsilon}}{\sqrt{G_{t-1} + \epsilon}} \cdot g_{t-1}.$$

Для полноты картины запишем полную версию Adadelta в псевдокоде:

```
g_cached = lmb * g_cached + (1 - lmb) * dx**2
u_cached = lmb * u_cached + (1 - lmb) * u**2
u = -np.sqrt(u_cached) * grad / (np.sqrt(g_cached) + eps)
theta += u
```

Следующий вариант, очень похожий на предыдущие, — это алгоритм RMSprop; RMS здесь означает root mean squares, среднеквадратическое отклонение (при чем оно здесь, мы увидим буквально в следующем абзаце). RMSprop и Adadelta — фактически близнецы-братья, хотя придуманы они были почти одновременно и независимо разными людьми. RMSProp при этом так никогда и не был опубликован, все ссылки на него так или иначе приводят к известному курсу Джеффри Хинтона по нейронным сетям [532], а статья об Adadelta [584] была опубликована в том же году, что и курс Хинтона. Такое совпадение неудивительно, ведь оба метода основаны на давно известной классической идее применения инерции, только на этот раз RMSprop использует ее для оптимизации метапараметра скорости обучения.

Основная разница между RMSprop и Adadelta состоит в том, что RMSprop не делает вторую поправку с изменением единиц и хранением истории самих обновлений, а просто использует корень из среднего от квадратов (вот он где, RMS) от градиентов:

$$u_t = -\frac{\eta}{\sqrt{G_{t-1}+\epsilon}} \cdot g_{t-1}.$$

Значение ϵ обычно берут равным 0,9, а $\eta = 0{,}001$.

И еще один адаптивный алгоритм оптимизации, который тоже в последнее время часто используется при обучении нейронных сетей — это Adam [278]. Как и предыдущие два метода, он является модификацией Adagrad, но использует сглаженные версии среднего и среднеквадратичного градиентов:

$$m_t = \beta_1 m + (1-\beta_1) g_t, \quad v_t = \beta_2 m + (1-\beta_2) g_t^2, \quad u_t = \frac{\eta}{\sqrt{v+\epsilon}} m_t.$$

В псевдокоде это тоже нетрудно отразить:

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
u = - learning_rate * m / (np.sqrt(v) + eps)
theta += u
```

В исходной статье [278] рекомендуются значения $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$, $\epsilon = 10^{-8}$.

На практике эффект от хорошего современного алгоритма оптимизации почувствовать очень легко; все хорошие алгоритмы оптимизации, разумеется, практически мгновенно попадают в библиотеки для обучения. Например, в TensorFlow для этого достаточно просто заменить `tf.train.GradientDescentOptimizer` на процедуру `tf.train.AdamOptimizer`:

```
train_op = tf.train.AdamOptimizer().minimize(loss)
```

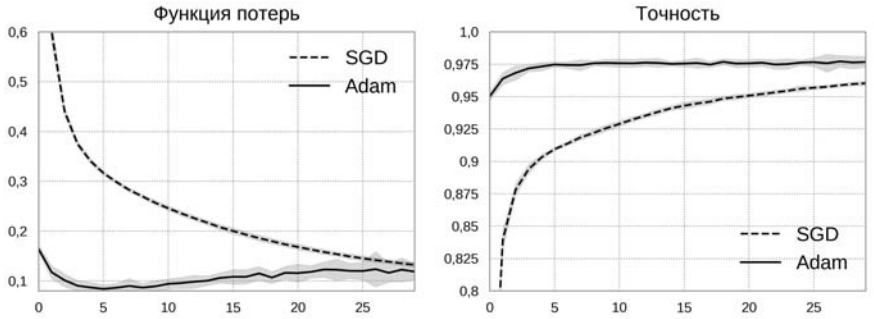


Рис. 4.6. Сравнение стохастического градиентного спуска и Adam

И результат не заставит себя ждать: графики, аналогичные рис. 4.2 и 4.5, для алгоритма Adam показаны на рис. 4.6. Очевидно, Adam сходится куда нужно гораздо быстрее (хоть и с большей дисперсией).

Что же выбрать? Как определиться, какой из многочисленных существующих методов оптимизации использовать, особенно учитывая, что никакой дополнительной работы они от вас не требуют (в наше время это вопрос передачи того или иного параметра в оптимизатор)? Это вопрос сложный, и в реальности конкретной задачи лучше всего будет попробовать сразу несколько и посмотреть, какие из них вообще сходятся, какие дают ошибку лучше и т. д.

Но некоторые общие советы дать можно (см. также [101, 458]). Если ваши данные достаточно разреженные, то вам точно стоит посмотреть в сторону адаптивных алгоритмов. RMSprop и Adadelta различаются не очень сильно, а Adam — это прямое расширение RMSprop, и, наверное, Adam будет в среднем наилучшим выбором на данный момент. Однако в нашей практике бывали задачи, на которых Adam расходился, а, скажем, Adadelta давала нормальные результаты, так что если вдруг Adam не работает, не отчаивайтесь раньше времени. Некоторые разработчики предпочитают использовать стохастический градиентный спуск с изменением скорости обучения по расписанию; он, конечно, в подавляющем большинстве случаев тоже приведет к решению, хоть и медленнее.

Важный практический совет состоит в том, что при градиентном спуске нужно следить за ошибкой на валидационном множестве и останавливать процесс в тот момент, когда ошибка начинает увеличиваться. Во многих случаях градиентный спуск может привести к оверфиттингу, упасть в слишком глубокий минимум функции ошибки, который плохо будет обобщаться на новые данные. Собственно, алгоритмам градиентного спуска неоткуда знать о том, насколько хорошо обобщается то, что они делают, они просто оптимизируют ошибку на тренировочном множестве. Поэтому мы сами должны следить за качеством обобщения, которое в простейшем случае соответствует ошибке на валидационном множестве.

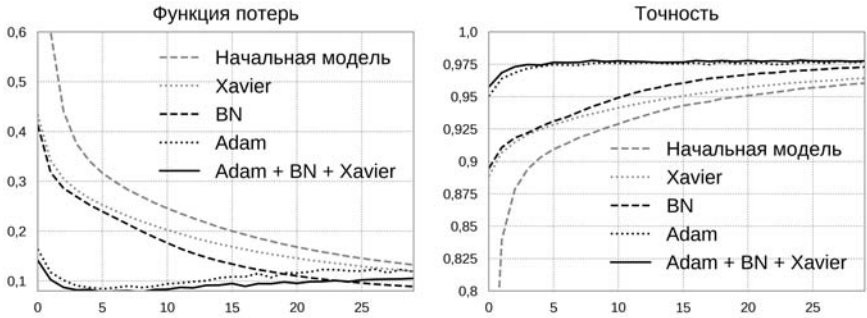


Рис. 4.7. Сравнение разных методов для улучшения обучения нейронных сетей

И здесь появляется еще одно замечание, которое может поначалу показаться философским: выходит, что мы использовали валидационное множество для оптимизации нашей модели, и множество это «запачкалось», его нельзя уже считать валидационным! Это кажется странным: что же, получается, мы «испортили» целый большой кусок данных только для того, чтобы выбрать шаг, на котором остановиться? Давайте тогда уж добавим его в тренировочный набор и заодно еще и дообучимся.

Тем не менее, такое использование валидационного множества может иметь смысл: его часто используют не только для того, чтобы выбрать шаг для остановки, но и для того, чтобы оптимизировать параметры алгоритма обучения и гиперпараметры модели. Поэтому в реальных системах данные часто разбиваются не на две части, а на три:

- *тренировочное* множество (training set) используют, как и раньше, чтобы обучать модель;
- *валидационное* множество (validation set) применяют, чтобы подгонять гиперпараметры, вовремя останавливать процесс обучения и т. д.;
- а третье, *тестовое* множество (production set) используют уже для окончательной оценки качества.

Эта схема хорошо зарекомендовала себя на практике, и мы тоже советуем ее использовать; впрочем, в примерах в этой книге, скорее всего, мы будем ограничиваться только тренировочным и тестовым множествами, потому что не будем специально подбирать гиперпараметры.

А закончим мы эту главу общим сравнением, показав тот самый «невидимый прогресс», во многом благодаря которому мы сейчас способны обучать огромные и сложные нейронные сети. На рис. 4.7 показаны графики функции ошибки и точности классификации на MNIST для всех тех методов, которые мы рассматривали в этой главе (снова среднее по 10 запускам, но на этот раз без дисперсий, чтобы не загромождать график). Кроме того, здесь показана еще одна, самая главная кривая

(сплошная на рис. 4.7), соответствующая аналогичной модели, к которой применены *все* эти нововведения: и инициализация Ксавье, и нормализация по минибатчам, и Adam в качестве алгоритма оптимизации. Как видите, новшества не поглощают друг друга: хотя самым важным оказывается переход от стохастического градиентного спуска к Adam, сплошная кривая все же идет заметно выше графика последнего. Конечно, для MNIST все это не так важно, и разница не столь велика, но она заметна даже здесь — а для больших датасетов она часто оказывается разницей между «можем обучить» и «не можем».

Теперь, когда мы вооружились самыми современными методами обучения, можно заняться и архитектурами. В следующей главе нас ждет одна из главных и самых популярных «нестандартных» архитектур: сверточные нейронные сети.

Глава 5
Сверточные нейронные сети и автокодировщики,

или Не верь глазам своим

TL;DR

Данная глава вводит важнейшие архитектуры сверточных нейронных сетей и автокодировщиков. В ней мы:

- начнем, как обычно, с биологии — со зрительной коры головного мозга;
- подробно рассмотрим, как работают сверточные сети;
- дадим обзор современных сверточных архитектур;
- перейдем к сетям, обучающимся без учителя, — автокодировщикам;
- после теоретического обзора закончим главу практическим сравнением разных типов автокодировщиков (в том числе сверточного) на наборе данных MNIST.

5.1. Зрительная кора головного мозга

> ...The visual system of the brain has the organization, computational profile, and architecture it has in order to facilitate the organism's thriving at the four Fs: feeding, fleeing, fighting, and reproduction.
>
> *P. S. Churchland, V. S. Ramachandran, T. J. Sejnowski. A Critique of Pure Vision [82]*

Сверточные нейронные сети (convolutional neural networks, CNN) — это весьма широкий класс архитектур, основная идея которых состоит в том, чтобы переиспользовать одни и те же части нейронной сети для работы с разными маленькими, локальными участками входов. Как и многие другие нейронные архитектуры, сверточные сети известны довольно давно, и в наши дни у них уже нашлось много самых разнообразных применений, но основным приложением, ради которого люди когда-то придумали сверточные сети, остается *обработка изображений*.

И это не случайно: идея сверточных сетей во многом мотивирована исследованиями о зрительной коре головного мозга. Так что в этой главе мы снова делаем шаг назад и начинаем с биологии. Конечно, изучение механизмов зрения — это очень большая тема, которой мы можем лишь слегка коснуться; в качестве достаточно популярных и интересных книг рекомендуем две доступные бесплатно [234, 562], одна из которых — популярное изложение классических исследований по теме от самого Дэвида Хьюбела[1] [235, 236, 569].

Люди давно задумывались о том, как именно мы видим окружающий мир, как работает зрение. Связь глаз со зрением была, простите за каламбур, очевидной даже древним[2]. Глаз как оптический прибор изучали Леонардо да Винчи, Иоганн Кеплер и многие другие великие физики, отмечавшие его выдающиеся оптические свойства. Впрочем, мнения разнились. Герман Гельмгольц писал так: «Какой плохой оптик Господь Бог! Я счел бы себя вправе самым резким образом выразиться о небрежности работы оптика и возвратить ему прибор с протестом, если бы он

---

[1] *Дэвид Хьюбел* (David Hunter Hubel, 1926–2013) — канадский нейрофизиолог, лауреат Нобелевской премии по физиологии и медицине 1981 года. Главным делом его научной жизни стало продолжавшееся более 20 лет сотрудничество с Торстеном Визелем (Thorsten Nils Wiesel) [237], в течение которого Хьюбел и Визель выяснили очень многое о структуре и функциях зрительной коры мозга. Проводя эксперименты на кошках и обезьянах, они впервые применили к исследованиям зрительной коры новую технологию микроэлектродов, позволяющих регистрировать возбуждение отдельных нейронов. С помощью таких электродов они изучили, как происходит бинокулярное зрение, а главное — выяснили, на что реагируют отдельные нейроны, как некоторые из них служат детекторами границ, другие реагируют на форму линейных сегментов и т. д. Словом, Хьюбел и Визель открыли многое из того, о чем мы говорим в этом разделе.

[2] Вообще говоря, назначение органов человеческого тела выяснить было совсем не просто; широко известно, например, мнение Аристотеля, который считал мозг своеобразным радиатором, охлаждающим кровь. Но о назначении глаз людям, слышавшим о печальной истории царя Эдипа, действительно догадаться несложно.

вздумал продать мне инструмент, обладающий такими недостатками, как человеческий глаз». Но все-таки на самом деле способности глаза к аккомодации, то есть изменению фокусного расстояния, и адаптации, то есть способности хорошо видеть при разных условиях освещенности, и впрямь потрясающе хороши.

Есть известная цитата Дарвина из «Происхождения видов», которую иногда цитируют так: «В высшей степени абсурдным, откровенно говоря, может показаться предположение, что путем естественного отбора мог образоваться глаз со всеми его неподражаемыми изобретениями...». Любопытно, что иногда эту цитату всерьез приводят креационисты как свидетельство того, будто бы Дарвин «сам понимал» ограниченность теории эволюции. Однако цитата буквально тут же продолжается так: «...Разум мне говорит: если можно показать существование многочисленных градаций от простого и несовершенного глаза к глазу сложному и совершенному, причем каждая ступень полезна для ее обладателя, а это не подлежит сомнению; если, далее, глаз когда-либо варьировал и вариации наследовались, а это также несомненно; если, наконец, подобные вариации могли оказаться полезными животному при переменах в условиях его жизни — в таком случае затруднение, возникающее при мысли об образовании сложного и совершенного глаза путем естественного отбора, хотя и непреодолимое для нашего воображения, не может быть признано опровергающим всю теорию». И действительно, последние исследования показывают, что сложные глаза, способные различать формы, образовались у разных животных в процессе эволюции совершенно независимо целых пятьдесят, а то и сто раз; а известная модель Нильссона и Пелгер [393] показала, что эволюция полноценного глаза от первых улавливающих свет клеток могла произойти очень быстро, в течение буквально нескольких тысяч поколений, или нескольких сотен тысяч лет. Но нас, конечно, в этой книге интересует не глаз, а то, что потом происходит с прочитанным с сетчатки изображением.

Зрительный нерв — толстый пучок аксонов ганглионарных клеток, по которому информация с сетчатки доходит до мозга, — отмечали еще средневековые анатомы, и его важность для зрения была очевидной. Зрительный нерв входит в таламус, отдел мозга, обрабатывающий информацию от органов чувств, там первичная обработка происходит в так называемом латеральном коленчатом теле (lateral geniculate nucleus, LGN), а затем зрительная информация от LGN поступает в собственно зрительную кору (visual cortex).

Все это было известно людям еще в XIX веке, но настоящие исследования зрительной коры и восприятия изображений не могли начаться раньше, чем люди смогли заглянуть на уровень отдельных нейронов, то есть в первой половине XX века. Об исследованиях человеческого зрения и обработки зрительных сигналов можно написать отдельную книгу (и таких книг, конечно же, написано немало). Есть масса очень интересных примеров зрительных иллюзий и странных заболеваний[1], но наша цель сейчас куда скромнее: мы попробуем отметить несколько важных черт в устройстве зрительной коры, которым можно найти некоторые

[1] Например, в нашем мозге почти наверняка есть особые области и особые пути обработки человеческих лиц, отдельные от распознавания всех остальных объектов. Как ученые пришли к этому выводу?

условные соответствия в архитектуре современных искусственных нейронных сетей. Подчеркнем на всякий случай, что речь идет исключительно о том, чтобы «вдохновляться идеями»: реальная работа мозга пока что остается для нас слишком сложной и непонятной, и смоделировать зрительную кору или даже отдельные ее части пока выше сил человеческих.

Зрительная кора, как ни странно, расположена сзади, в затылочной доле головного мозга. Она делится на несколько частей, которые обычно незамысловато называются *зрительная зона V1* (visual area one), которую также называют *стриарной корой* или *первичной зрительной корой*, *зрительная зона V2* (visual area two), зрительная зона V3 и т. д., до V6 и V7. Зоны отличаются друг от друга физиологией, архитектурой да и просто обособленным положением в коре, и исследователи не сомневаются, что они различаются и по своим функциям. Хотя на самом деле функциональная специализация зон пока что до конца не ясна, понятно, что функции зон зрительной коры становятся постепенно все более и более общими. По нынешним представлениям:

- в зоне V1 выделяются локальные признаки небольших участков считанного с сетчатки изображения; это для нас сейчас самое интересное, и об этом мы подробно поговорим ниже;
- V2 продолжает выделять локальные признаки, слегка обобщая их и добавляя бинокулярное зрение (то есть стереоэффект от двух глаз);
- в зоне V3 распознается цвет, текстуры объектов, появляются первые результаты их сегментации и группировки;
- зона V4 уже начинает распознавать геометрические фигуры и очертания объектов, пока несложных; кроме того, именно здесь наиболее сильна модуляция посредством нашего *внимания*: активация нейронов в V4 не равномерна по всему полю зрения, а сильно зависит от того, на что мы осознанно или неосознанно обращаем внимание;
- зона V5 в основном занимается распознаванием движений, пытаясь понять, куда и с какой скоростью передвигаются в зоне видимости те самые объекты, очертания которых выделились в зоне V4;
- в зоне V6 обобщаются данные о всей картинке, она реагирует на изменения по всему полю зрения (wide-field stimulation) и изменения в картинке вследствие того, что передвигается сам человек;
- иногда также выделяют зону V7, где происходит распознавание сложных объектов, в частности человеческих лиц.

Такая функциональная специализация — это первое замечание о зрительной коре, которое хорошо соответствует тому, что мы обычно видим в глубоких нейронных сетях: более высокие уровни нужны для того, чтобы выделять более общие

Очень просто: при определенных повреждениях мозга (например, инсультах) вырабатывается так называемая *предметная агнозия*, когда человек не может распознавать и правильно идентифицировать объекты... но с распознаванием лиц у него при этом все в порядке! А распознавание лиц отключается при *лицевой агнозии*, которая, в свою очередь, совершенно не мешает распознавать другие объекты.

признаки, соответствующие абстрактным свойствам входа, а на нижних уровнях признаки более конкретные. В сверточных сетях, разумеется, эффект будет тот же самый. Но есть и другие интересные свойства архитектуры зрительной коры, которые нашли отражение в машинном обучении.

Кроме зон V1–V7, выделяют и другие области, а среди перечисленных иерархия не такая уж строгая: есть масса прямых связей, когда, например, нейроны из зоны V1 подаются на вход не только в зону V2, но и напрямую в зону V5; это тоже найдет отражение в сверточных архитектурах.

Кроме того, в отличие от большинства архитектур искусственных нейронных сетей, между нейронами в мозге всегда присутствует очень сильная обратная связь от более высоких уровней к более низким. Например, современные исследования предполагают, что внимание, которое зарождается в зоне V4, потом переходит обратно к V2 и V1 (там тоже присутствуют сильная модуляция на основе внимания!) [19].

В искусственных нейронных сетях тоже вводят подобные механизмы внимания, помогающие выбрать, какие именно входы и промежуточные выходы сети сейчас нужно учитывать больше всего — о них мы поговорим в разделе 8.1. Но все-таки именно обратную связь мы, специалисты по машинному обучению, пока не очень умеем «готовить»; придумать математическую модель для того, как мозг с ней управляется — одна из важнейших открытых задач нейробиологии.

Кстати, по современным представлениям выходы нейронов из зоны V1 и V2 обрабатываются сразу двумя параллельными путями: *вентральный* путь из V2 направляется к V4 отвечает на вопрос «Что?» (его иногда называют *What Pathway*), отвечая за распознавание форм и объектов, а *дорсальный* путь (*Where Pathway*) отвечает на вопрос «Где?»[1], проходит из V2 к V5 и V6 и занимается в основном распознаванием движений, управлением движением глаз и рук (особенно если речь идет о том, как достать находящийся в поле зрения объект) [544]. Настолько детальной специализацией мы наши искусственные сети снабдить не сможем, но эта идея, вообще говоря, тоже регулярно используется: часто есть смысл обработать один и тот же вход несколькими разными способами и только потом объединить результаты, скажем, суммированием или конкатенацией. В сверточных сетях так часто делают, когда обрабатывают вход с помощью сразу нескольких размеров фильтров; о фильтрах речь пойдет буквально в следующем разделе.

Но и это еще не все. Из всей зрительной коры наиболее хорошо изучена зона V1, первичная зрительная кора: она ближе всего к собственно входам, достаточно просто устроена, и там проще понять, за что «отвечают» отдельные нейроны и как они друг с другом взаимодействуют [71]; именно с зоной V1 были связаны самые громкие результаты Хьюбела и Визеля. Оказалось, что зона V1 сама состоит из шести уровней нейронов: входные сигналы приходят из латерального коленчатого тела в четвертый уровень, дальнейшие сигналы выходят из уровней 2 и 3, а обратная

[1] К сожалению, When Pathway в мозге пока не нашелся...

связь — из шестого. Нейроны в зоне V1 располагаются не случайно: зона V1 содержит полную «карту» полей зрения обоих глаз, то есть близкие участки сетчатки обрабатываются близкими нейронами в V1. Любопытно, что при этом «картировании» локальная структура переносится очень точно, а вот на глобальном уровне есть серьезные искажения: во-первых, центральный участок поля зрения сильно увеличен (половина всех нейронов отвечают за 2 % поля зрения), во-вторых, есть геометрические искажения, похожие на использование полярных координат: концентрические круги и радиальные линии на картинке преобразуются в вертикальные и горизонтальные линии в V1. Это позволяет сохранять инвариантность изображения при смене нашей позиции и угла зрения. Такие преобразования мы применять не будем, но давайте запомним, что каждый нейрон в V1 работает с очень маленьким участком изображения (он называется *рецептивным полем*, receptive field), и между ними сохраняются пространственные взаимосвязи, подобные исходной картинке.

Интересно понять, на что, собственно, реагируют нейроны в V1. Кроме расположения, отраженного в «карте», разные нейроны в V1 распознают:

- *ориентацию*, то есть нейрон реагирует, например, на то, что освещенность вдоль диагонали изображения высокая, а по двум другим углам низкая; или на то, что освещенность нижней части изображения выше, чем верхней; таким образом нейроны распознают границы изображений (edge detection);
- *пространственную частоту*, то есть то, насколько часто меняется освещенность в пределах рецептивного поля нейрона;
- *направление*: в отличие от большинства сетей, которые мы будем рассматривать ниже, человеку интересна не только и не столько статичная картинка, сколько происходящие в ней движения; нейроны умеют «запоминать» и сравнивать предыдущие входы с последующими, распознавая таким образом направление не только в пространстве, но и во времени; аналогично распознается и *временная частота*;
- *различие между глазами*: в V1 у каждого нейрона два рецептивных поля, для каждого глаза, и хотя большинство нейронов в основном «посвящены» одному конкретному глазу (то есть не реагируют на стимулы с другого), некоторые распознают как раз различия между тем, что видят левый и правый глаз;
- *цвет*, относительно которого нейроны обычно распознают одно из трех основных направлений: красный — зеленый, синий — желтый и черный — белый.

Математическое описание того, как нейроны зрительной коры активируются в зависимости от ориентации [104, 348], оказалось тесно связано с так называемыми *фильтрами Габора* [168] — видом математических преобразований, очень эффективным для выделения границ объектов на изображениях (edge detection). А знаменитый набор признаков для изображений SIFT (scale-invariant feature transform, масштабно-инвариантное преобразование признаков), разработанный Дэвидом Лоу в конце 1990-х годов [334, 335], во многом соответствует тому, как

активируются нейроны зрительной коры. Сегодня развитие глубоких сетей привело к тому, что сейчас мы получаем примерно те же (только еще лучше работающие) признаки на нижних уровнях сверточных сетей — их примеры мы еще не раз увидим в этой главе.

Рассказ о зоне V1 плавно подводит нас к последнему замечанию: как видно, обработка изображений в мозге устроена как весьма глубокая нейронная сеть. В частности, уже в самых ранних работах Хьюбела и Визеля клетки зрительной коры (а именно зоны V1) подразделялись на *простые* и *сложные*. Простые клетки реагируют только на участки и полосы света, края стимулов, проходящих в определенном месте под определенным углом. А сложные клетки могут, например, реагировать на ориентацию независимо от конкретного положения сигнала. Достигается это тем, что сложные клетки относятся к следующему уровню обработки, получая на вход результат активации простых клеток; поэтому их рецептивное поле обычно немного больше, а выявляемые ими признаки оказываются инвариантными к местоположению сигнала. В частности, клетки, реагирующие на движение сигнала, тоже всегда являются сложными по данной классификации.

И это только начало: в одной только зоне V1 выделяют шесть уровней, а ведь сигнал по ходу движения проходит через сразу несколько зон, начиная с латерального коленчатого тела (в котором, кстати, тоже шесть уровней) и заканчивая высокоуровневыми зонами V6 и V7. Это тоже нашло отражение в искусственных нейронных сетях: сверточные сети для обработки изображений — это самые глубокие из существующих сетей. С помощью идей, которые мы изложим в этой главе, можно обучить сеть в несколько десятков и даже сотен уровней!

Но хватит биологии. Начиная со следующего раздела мы переходим собственно к изложению идей и математики сверточных сетей. Мы увидим, как с помощью простой идеи переиспользования одних и тех же весов для разных участков входного изображения можно радикально сократить число параметров в сети, ничего не потеряв в выразительности. Сверточные сети — одна из самых популярных архитектур современных нейронных сетей, и эта глава — одна из центральных в книге.

5.2. Свертки и сверточные сети

> — Теперь, если ты немножко распустишь шнурки, которыми стянут мой верхний щит, я посмотрю, не удастся ли мне свернуться в шар. Это может оказаться полезным.
>
> *Р. Киплинг. Как появились броненосцы*

Итак, в предыдущем разделе мы обсудили некоторые особенности архитектуры зрительной коры. Как мы выяснили, именно зрительная кора, а точнее, исследования Хьюбела и Визеля на рубеже 50–60-х годов XX века [235, 569], мотивировали многие идеи, сегодня использующиеся в глубоких архитектурах.

И здесь снова появляется один из лейтмотивов нашей книги, по крайней мере ее исторических экскурсов: идея сверточных сетей появилась по меркам машинного обучения очень давно. Можно сказать, что первой настоящей сверточной сетью, позаимствовавшей для информатики воплощенные природой в зрительной коре идеи, был Neocognitron Кунихико Фукусимы, появившийся в 1979–1980 годах [166, 167][1]. Впрочем, Фукусима не использовал градиентный спуск и вообще обучение с учителем, а его работы были довольно прочно забыты. Снова сверточные сети в уже вполне современной форме появились только в работах группы Яна ЛеКуна в конце 1980-х годов [18, 188, 205], и с тех пор и до наших дней они вполне успешно применяются для распознавания изображений и многих других задач (см., например, свежие обзоры [438, 522]).

Но что же, собственно, такое в данном случае *свертка* и какое она имеет отношение к нейронным сетям? Чтобы ответить на этот вопрос, давайте сначала отступим на шаг назад. Краеугольным камнем всех нейронных сетей являются *аффинные преобразования*. В каждом слое полносвязной сети повторяется одна и та же операция: на вход подается вектор, который умножается на матрицу весов, а к результату добавляется вектор свободных членов; только после этого к результату применяется некая нелинейная функция активации. И во всех сетях, которые мы до сих пор строили, такой подход использовался постоянно, независимо от структуры или происхождения данных. Будь то изображения, текст или музыка, мы вновь и вновь применяем аффинное преобразование в каждом слое нашей сети, предварительно приведя данные к векторной форме.

Однако многие типы данных имеют свою собственную внутреннюю структуру, которая отлично известна нам заранее. Главный пример такой структуры в этой главе — изображение, которое обычно представляют как массив векторов чисел: если изображение черно-белое, то это просто массив интенсивностей, а если цветное, то массив векторов из трех чисел, обозначающих интенсивности трех основных цветов (красного, зеленого и черного в стандартном RGB, синего, зеленого и красного в трех типах колбочек в человеческом глазе и т. д.). Если же обобщить такую внутреннюю структуру до максимальной все еще полезной нам общности, описание получится такое:

1) исходные данные представляют собой многомерный массив («тензор»);
2) среди размерностей этого массива есть одна или более осей, порядок вдоль которых играет важную роль; например, это может быть расположение пикселов в изображении, временная шкала для музыкального произведения, порядок слов или символов в тексте;
3) другие оси обозначают «каналы», описывающие свойства каждого элемента по предыдущему подмножеству осей; например, три компонента для изображений, два компонента (правый и левый) для стереозвука и т. д.

[1] Мы уже вспоминали об этой модели в разделе 3.3, потому что именно в Neocognitron впервые появились перцептроны с функцией активации, похожей на ReLU.

Когда мы обучаем полносвязные нейронные сети, это дополнительное знание о структуре задачи никак не используется. Вспомним пример сети для распознавания рукописных цифр, которую мы строили в разделе 3.6: там мы просто превращали изображение размера 28×28 пикселов в вектор длины 784 и подавали его на вход. Получалось, что наши аффинные преобразования никак не учитывали структуру картинки, топологию данных!

Но ведь она не просто присутствует, а играет определяющую роль: конечно же, взаимное расположение пикселов в картинке важно для распознавания цифр... Получается, что сети приходилось на основе интенсивностей отдельных пикселов выучить не только то, какая форма соответствует какой цифре, но заодно и вообще понятие формы, тот факт, что некоторые компоненты входного вектора представляют собой соседние пикселы, а значит, они сильно скоррелированы, и так далее... непростую задачу мы предлагаем нашей несчастной полносвязной сети.

Если мы не будем делать вид, что ничего не знаем о структуре входов (разделении цветовых каналов изображения, порядке нуклеотидов в ДНК и т. д.), а станем ее напрямую использовать, это может существенно помочь нам в обучении нейронных сетей. В этом разделе мы рассмотрим такой подход на примере сверточных слоев для задач компьютерного зрения. Мы обсудим очень важную часть современных глубоких нейронных сетей и снова увидим, как очень простая концепция приводит к фантастическим результатам.

Основная идея сверточной сети состоит в том, что обработка участка изображения очень часто должна происходить независимо от конкретного расположения этого участка. Грубо говоря, если вы хотите узнать на фотографии своего друга Васю, совершенно не важно, на 100 или на 200 пикселов ухо Васи отстоит от левого края фотографии. Узнать Васю можно было бы и на сильно обрезанной фотографии, где нет ничего, кроме его лица; это локальная задача, которую можно решать локальными средствами. Конечно, взаимное расположение объектов играет важную роль, но сначала их нужно в любом случае распознать, и это распознавание — локально и независимо от конкретного положения участка с объектом внутри большой картинки.

Поэтому сверточная сеть попросту делает это предположение в явном виде: давайте покроем вход небольшими окнами (скажем, 5×5 пикселов) и будем выделять признаки в каждом таком окне небольшой нейронной сетью. Причем — и тут ключевое соображение — признаки будем выделять в каждом окне одни и те же, то есть маленькая нейронная сеть будет всего одна, входов у нее будет всего $5 \times 5 = 25$, а из каждой картинки для нее может получиться очень много разных входов.

Затем результаты этой нейронной сети опять можно будет представить в виде «картинки», заменяя окна 5×5 на их центральные пикселы, и на ней можно будет применить второй сверточный слой, с уже другой маленькой нейронной сетью, и т. д. Скоро мы увидим, что в каждом сверточном слое будет совсем немного свободных параметров, особенно по сравнению с полносвязными аналогами.

Прежде чем переходить непосредственно к формальным определениям операции свертки, давайте разберемся с понятием канала в изображении. Обычно цветные картинки, подающиеся на вход нейронной сети, представлены в виде нескольких прямоугольных матриц, каждая из которых задает уровень одного из цветовых каналов в каждом пикселе изображения. Картинка размером 200×200 пикселов — это на самом деле 120 000 чисел, три матрицы интенсивностей размером 200×200 каждая. Если изображение черно-белое, как, например, в MNIST, то такая матрица будет одна. А если это не простая картинка, а, скажем, результат изображающей масс-спектрометрии, когда в каждом пикселе находится целый спектр, то матриц может быть очень много. Но в любом случае мы будем предполагать, что в каждом пикселе входного изображения стоит некоторый тензор (обычно одномерный, то есть вектор чисел), и его компоненты называются *каналами* (channels).

Такие же матрицы будут получаться и после сверточного слоя: в них попрежнему будет пространственная структура, соответствующая исходной картинке (но не в точности такая же — скоро об этом поговорим), однако каналов теперь может стать больше. Значения каждого признака, которые мы выделили из окон в исходном изображении, теперь будут представлять собой целую матрицу. Каждая такая матрица называется *картой признаков* (feature map). В принципе, каналы исходного изображения можно тоже называть картами признаков; аналогично мы часто будем называть карты признаков очередного слоя каналами.

Теперь осталось только формально определить, что же такое свертка и как устроены слои сверточной сети. Свертка — это всего лишь линейное преобразование входных данных особого вида. Если $\boldsymbol{x}^l$ — карта признаков в слое под номером l, то результат двумерной свертки с ядром размера $2d+1$ и матрицей весов W размера $(2d+1) \times (2d+1)$ на следующем слое будет таким:

$$y^l_{i,j} = \sum_{-d<=a,b<=d} W_{a,b} x^l_{i+a,j+b},$$

где $y^l_{i,j}$ — результат свертки на уровне l, а $x^l_{i,j}$ — ее вход, то есть выход всего предыдущего слоя. Иначе говоря, чтобы получить компоненту (i,j) следующего уровня, мы применяем линейное преобразование к квадратному окну предыдущего уровня, то есть скалярно умножаем пикселы из окна на вектор свертки. Это проиллюстрировано на рис. 5.1: мы применяем свертку с матрицей весов W размера 3×3 к матрице X размера 5×5. Обратите внимание, что умножение подматрицы исходной матрицы X, соответствующей окну, и матрицы весов W — это не умножение матриц, а просто скалярное произведение соответствующих векторов. А всего окно умещается в матрице X девять раз, и получается

$$\begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 4 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 2 & 3 & 1 & 0 \\ 0 & 4 & 3 & 2 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 9 & 5 & 4 \\ 8 & 8 & 10 \\ 8 & 15 & 12 \end{pmatrix}.$$

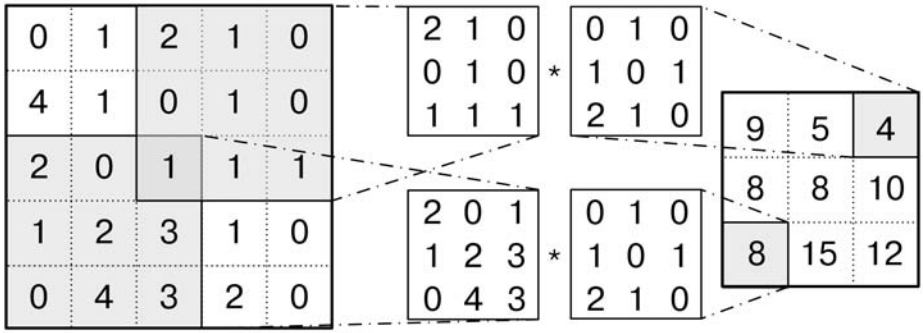


Рис. 5.1. Пример подсчета результата свертки: два примера подматрицы и общий результат

Здесь мы обозначили свертку карты признаков X с помощью матрицы весов W через $X * W$, как принято обозначать свертку в математике.

Если размер свертки будет выражаться четным числом, то в одном из случаев в пределах суммирования неравенство станет строгим; здесь больше нет «естественного» выбора для центра окна, и его выбор из четырех вариантов может зависеть от реализации в конкретной библиотеке.

Это преобразование обладает как раз теми свойствами, о которых мы говорили выше:

- свертка сохраняет структуру входа (порядок в одномерном случае, взаимное расположение пикселов в двумерном и т. д.), так как применяется к каждому участку входных данных в отдельности;
- операция свертки обладает свойством разреженности, так как значение каждого нейрона очередного слоя зависит только от небольшой доли входных нейронов (а, например, в полносвязной нейронной сети каждый нейрон зависел бы от всех нейронов предыдущего слоя);
- свертка многократно переиспользует одни и те же веса, так как они повторно применяются к различным участкам входа.

Почти всегда после свертки в нейронной сети следует нелинейность, которую мы можем записать так:

$$z^l_{i,j} = h(y^l_{i,j}).$$

В качестве функции h часто используют ReLU, особенно в очень глубоких сетях, но и классические σ и tanh тоже встречаются. Подробно останавливаться на типах нелинейностей, использующихся в сверточных сетях, мы сейчас не будем; наша первоочередная задача — сформировать интуиции по поводу сверточных сетей.

Прежде чем двигаться дальше, продемонстрируем, как свертки работают на практике, на численном примере в TensorFlow. Начнем, как обычно, с импортирования TensorFlow и `numpy`:

```
import tensorflow as tf
import numpy as np
```

Поскольку мы никакую модель обучать не планируем, в качестве переменных будет достаточно определить «заглушки» заданного размера:

```
x_inp = tf.placeholder(tf.float32, [5, 5])
w_inp = tf.placeholder(tf.float32, [3, 3])
```

Давайте теперь создадим сверточный слой на TensorFlow. Все, что нужно знать для этого, мы уже знаем: сверточный слой должен брать скользящие окна из исходного изображения и применять к ним одни и те же веса. Свертка у нас двумерная, то есть «скользящее окно» — это квадратик размером в несколько пикселов; осталось только разобраться, как это задать в коде.

Чтобы определить свертку, сначала нужно разобраться с размерностями тензоров. В TensorFlow двумерные свертки реализуются с помощью функции `conv2d`, которая принимает на вход тензор сверточных весов и тензор карт признаков набора изображений... ох, звучит как кеннинг[1]. Дело в том, что входные данные для двумерной свертки в TensorFlow должны иметь четырехмерную структуру, которая выглядит так:

$$[\text{размер батча, высота, ширина, каналы}].$$

Например, если мы используем мини-батчи размером 32 изображения в каждом и обучаем сеть на RGB-изображениях лиц размером 28×28 пикселов, то итоговая размерность тензора данных будет $[32, 28, 28, 3]$: если перемножить все размерности, получится, что каждый мини-батч, подающийся на вход сети, содержит около 75 тысяч чисел! Каждое изображение при этом представлено $28 \times 28 \times 3 = 2352$ вещественными числами.

А размерность тензора сверточных весов определяется размерами ядра свертки и числом каналов как на входе, так и на выходе; получается снова четырехмерный тензор, который на этот раз нужно интерпретировать следующим образом:

$$[\text{высота, ширина, входные каналы, выходные каналы}].$$

[1] В скальдической поэзии использовали стандартные метафоры, которые по-русски передаются существительными в родительном падеже. Например, «перина дракона» — это золото, сокровища, «лебедь крови» — ворон. Интересно в кеннингах то, что их можно применять рекурсивно, разворачивая части кеннинга с помощью других кеннингов: «шип битвы» — это меч, «пот шипа битвы» — кровь, «лебедь пота шипа битвы» — ворон. Встречаются даже рекурсивные кеннинги: «буря шипов битвы» — это снова битва.

Например, для фильтра размером 3×3, применяемого к тому же самому трехканальному изображению и дающему на выходе 32 карты признаков, мы получим тензор размерности $[3, 3, 3, 32]$, в нем будет всего $3 \times 3 \times 3 \times 32 = 288$ весов. А на выходе эти веса, если мы будем предполагать, что в картинку 28×28 пикселов помещается 26×26 окон 3×3, превратят входную картинку в $26 \times 26 \times 32 = 21\,632$ числа, а это значит, что в полносвязной сети для аналогичной операции потребовалась бы матрица размером $28 \times 28 \times 21\,632 = 16\,959\,488$ весов — разница в пятьдесят тысяч раз!

Конечно, эта разница получается исключительно из-за того, что мы переиспользовали одни и те же веса много раз. В реальности выходит, что сама форма операции свертки служит очень сильным регуляризатором, который выражает идею о том, что выделение локальных признаков в изображении не должно зависеть от конкретного места, где эти признаки располагаются. И, безусловно, такая огромная разница в требующемся числе весов приводит к тому, что процесс обучения сети тоже сильно упрощается.

На этом месте читателю может показаться, что и качество итоговой модели в случае использования сверточных слоев может пострадать по сравнению с полносвязными слоями: в конце концов, много параметров — это хорошо? Однако, как показывает практика, верно прямо обратное: сверточные сети почти всегда оказываются существенно лучше полносвязных в задачах, связанных с компьютерным зрением и, вообще говоря, обработкой входов, для которых выполняется основное предположение сверточных сетей о «локальности»: входы расположены в виде сетки той или иной размерности, и признаки нужно выделять из небольших подмножеств входов, расположенных близко в этой сети.

Впрочем, интуиция здесь тоже довольно понятная. Заменяя полносвязный слой сверточным, мы убиваем сразу двух зайцев:

- добавляем в явном виде предположение о том, что признаки нужно выделять локально, а также явно добавляем «геометрию» входа — сети больше не нужно самостоятельно выучивать эту геометрию;
- для локальной сети, которая призвана выражать эти локальные признаки, резко увеличиваем объем данных на входе: теперь для нее фактически каждая картинка превращается в кучу маленьких тренировочных примеров.

Осталась только одна проблема: один сверточный слой никак не сможет выразить взаимосвязь между пикселами, расположенными далеко друг от друга. Это действительно так, и чтобы решить эту проблему, мы будем строить глубокие сверточные сети; но об этом чуть позже, а пока вернемся к коду.

Итак, операция свертки в TensorFlow оперирует четырехмерными тензорами, а не обычными матрицами, поэтому входные данные нужно привести к соответствующим размерностям:

```
x = tf.reshape(x_inp, [1, 5, 5, 1])
w = tf.reshape(w_inp, [3, 3, 1, 1])
```

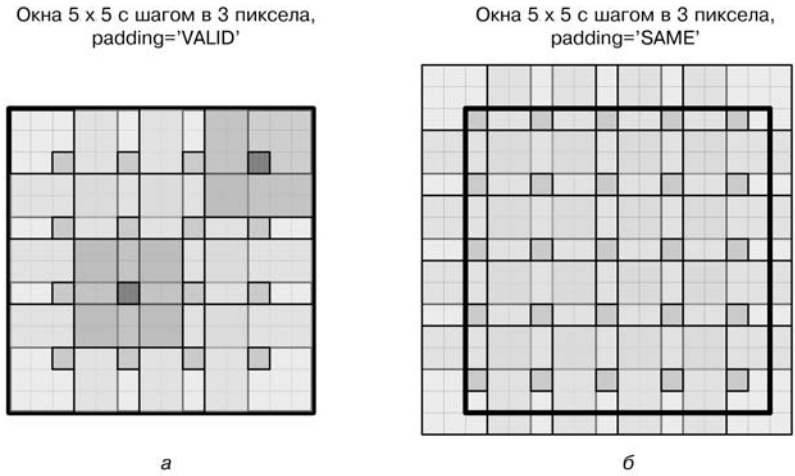


Рис. 5.2. Примеры расположения двумерных сверточных окон с центрами в каждом третьем пикселе по обеим осям: *а* — padding="VALID", все окна должны уместиться внутри входного массива; *б* — padding="SAME", окна могут выходить за рамки массива

Напомним, что первая размерность тензора `x` обозначает количество изображений в мини-батче, а последняя — число каналов изображения. Так как мы всего лишь хотим проиллюстрировать описанную выше теорию, нам будет достаточно одной черно-белой картинки. Теперь можно задавать саму операцию свертки:

```
x_valid = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding="VALID")
x_same = tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding="SAME")
x_valid_half = tf.nn.conv2d(x, w, strides=[1, 2, 2, 1], padding="VALID")
x_same_half = tf.nn.conv2d(x, w, strides=[1, 2, 2, 1], padding="SAME")
```

Как легко догадаться, `tf.nn.conv2d` создает сверточный слой, в качестве первых двух параметров получая на вход изображение и фильтр. Аргумент `padding` говорит, как быть с окнами, которые «вылезают» за границы входного массива. Этот аргумент может принимать два разных значения (рис. 5.2):

- `padding="VALID"` приведет к тому, что будут применяться свертки только на тех окнах, которые полностью помещаются в пределах входного массива; это значит, что размер массива на выходе станет меньше, чем был на входе; например, в нашу картинку размером 28×28 поместятся только 24×24 окон размера 5×5, а два пиксела «рамки» не смогут стать центрами сверточных фильтров, и следующий слой будет иметь размер 24×24, а не 28×28 (рис. 5.2, *а*);
- `padding="SAME"` приведет к тому, что размер слоя сохранится, а для выходящих за границы массива мы просто дополним входной массив нулями; теперь на выходе размер изображения не изменится, а пикселы из «рамки» тоже станут центрами сверточных фильтров, просто некоторые значения у них будут заведомо нулевыми (рис. 5.2, *б*).

Соответственно, в нашем численном примере для `padding="SAME"` получится:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 3 & 1 & 0 & 0 \\ 0 & 0 & 4 & 3 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 5 & 11 & 4 & 3 & 3 \\ 3 & 9 & 5 & 4 & 4 \\ 5 & 8 & 8 & 10 & 3 \\ 4 & 8 & 15 & 12 & 6 \\ 5 & 5 & 9 & 4 & 2 \end{pmatrix}.$$

Аргумент `strides` задает шаг по изображению: например, если бы мы подали на вход `strides=[1,3,3,1]`, окна были бы не все возможные, а с интервалом в три пиксела, как показано на рис. 5.2. Штриховка показывает, какие пикселы являются центрами окон, а на рис. 5.2, *а* два окна для примера выделены дополнительной штриховкой.

Обратите внимание, что параметр `strides` просто определяет, как часто мы применяем фильтры по каждой размерности входного тензора, а размерностей этих у него в данном случае четыре. Поэтому, например, первая компонента `strides` соответствует разным примерам в мини-батче, и если бы она была не равна единице, мы бы просто пропускали часть входных примеров. В данном случае это выглядит странно, и для функции `tf.nn.conv2d` вряд ли есть смысл задавать `strides` с неединичной первой компонентой (или пропускать часть цветовых фильтров в четвертой компоненте), но аргумент `strides` в TensorFlow является более общим и может быть применен к любому тензору, отсюда и «лишние» размерности (мы увидим их пользу чуть ниже, когда будем обсуждать операцию субдискретизации).

В нашем примере, меняя параметр `strides`, мы будем просто получать некоторые подматрицы тех матриц, которые мы вычисляли выше. Например, для `strides=[1, 2, 2, 1]` мы будем пропускать каждую вторую размерность и по строкам, и по столбцам; результат свертки с `padding="VALID"` будет такой:

$$\begin{pmatrix} 9 & 4 \\ 8 & 12 \end{pmatrix},$$

а с `padding="SAME"` получится следующее:

$$\begin{pmatrix} 5 & 4 & 3 \\ 5 & 8 & 3 \\ 5 & 9 & 2 \end{pmatrix}.$$

Итак, наша «модель» задана. Запишем входные данные:

```
x = np.array([[0, 1, 2, 1, 0],
              [4, 1, 0, 1, 0],
              [2, 0, 1, 1, 1],
              [1, 2, 3, 1, 0],
```

```
            [0, 4, 3, 2, 0]])
w = np.array([[0, 1, 0],
              [1, 0, 1],
              [2, 1, 0]])
```

Осталось только объявить сессию и вычислить результат:

```
sess = tf.Session()
y_valid, y_same, y_valid_half, y_same_half = sess.run(
    [x_valid, x_same, x_valid_half, x_same_half],
    feed_dict={x_inp: x, w_inp: w}
)
```

Давайте для проверки выведем то, что у нас получилось; поскольку на выходе у нас по-прежнему будут получаться четырехмерные тензоры, а мы хотели бы увидеть обыкновенные матрицы, некоторые (тривиальные) размерности мы зафиксируем нулями:

```
print "padding=VALID:\n", y_valid[0, :, :, 0]
print "padding=SAME:\n", y_same[0, :, :, 0]
print "padding=VALID, stride 2:\n", y_valid_half[0, :, :, 0]
print "padding=SAME, stride 2:\n", y_same_half[0, :, :, 0]
```

И теперь на выходе получаются вполне ожидаемые результаты:

```
padding=VALID:
[[  9.   5.   4.]
 [  8.   8.  10.]
 [  8.  15.  12.]]
padding=SAME:
[[  5.  11.   4.   3.   3.]
 [  3.   9.   5.   4.   4.]
 [  5.   8.   8.  10.   3.]
 [  4.   8.  15.  12.   6.]
 [  5.   5.   9.   4.   2.]]
padding=VALID, stride 2:
[[  9.   4.]
 [  8.  12.]]
padding=SAME, stride 2:
[[ 5.  4.  3.]
 [ 5.  8.  3.]
 [ 5.  9.  2.]]
```

Исторически первые идеи, похожие на современные свертки, появились в уже упоминавшейся модели Neocognitron, а уже в практически современном виде они были применены в конце 80-х годов XX века. Основным это направление стало для группы Яна ЛеКуна [18, 205]. Но любопытно, что почти одновременно практически такие же модели были разработаны группой японских исследователей под

руководством Вея Чжана [411, 492]; их работа получила название «нейронная сеть, инвариантная к сдвигу» (shift-invariant neural network) и тоже неоднократно применялась к распознаванию образов [243].

Итак, мы научились делать операцию свертки. После нее можно, как мы уже говорили, применить ту или иную нелинейную функцию h: она будет просто применяться к каждому элементу полученного тензора по отдельности. Но это еще не все. В классическом сверточном слое, кроме линейной свертки и следующей за ней нелинейности, есть и еще одна операция: *субдискретизация* (pooling; по-русски ее иногда называют еще операцией «подвыборки», от альтернативного английского термина subsampling; встречается и слово «пулинг», но добуквенных калек[1] в нашей книге, пожалуй, и так уже многовато).

Смысл субдискретизации прост: в сверточных сетях обычно исходят из предположения, что наличие или отсутствие того или иного признака гораздо важнее, чем его точные координаты. Например, при распознавании лиц сверточной сетью нам гораздо важнее понять, есть ли на фотографии лицо и чье, чем узнать, с какого конкретно пиксела оно начинается и в каком заканчивается. Поэтому можно позволить себе «обобщить» выделяемые признаки, потеряв часть информации об их местоположении, но зато сократив размерность.

Обычно в качестве операции субдискретизации к каждой локальной группе нейронов применяется операция взятия максимума (max-pooling); такая субдискретизация опять восходит еще к работам Хьюбела и Визеля, и похоже, что нейроны в зрительной коре поступают именно так. Иногда встречаются и другие операции субдискретизации; сверточные сети известны очень давно, и характерно, что первые конструкции группы ЛеКуна использовали для субдискретизации взятие среднего [18, 205], а не максимума. Исследования, в которых проводилось сравнение, обычно выступали в пользу max-pooling [306, 471]. А в работе [57] подробное сравнение разных операций субдискретизации привело к тому, что в качестве оптимальной альтернативы авторы предложили промежуточный вариант между максимумом и средним: не вдаваясь в детали, можно сказать, что они предложили брать максимум не по всему окну, а по некоторому его случайному подмножеству.

Однако именно максимум встречается на практике чаще всего и для большинства практических задач дает хорошие результаты, поэтому дальше, когда мы будем говорить просто о «субдискретизации», мы будем иметь в виду именно max-pooling, и формально станем определять субдискретизацию (в тех же обозначениях, что выше) так:

$$x_{i,j}^{l+1} = \max_{-d \le a \le d,\, -d \le b \le d} z_{i+a,\, j+b}^{l}.$$

[1] Ударение здесь на первый слог, но на самом деле оно вполне могло бы стоять и на втором... К сожалению, русскоязычная терминология — это пока очень больной вопрос для современного машинного обучения, и вряд ли он когда-нибудь решится окончательно, ведь прибывают все новые англоязычные термины.

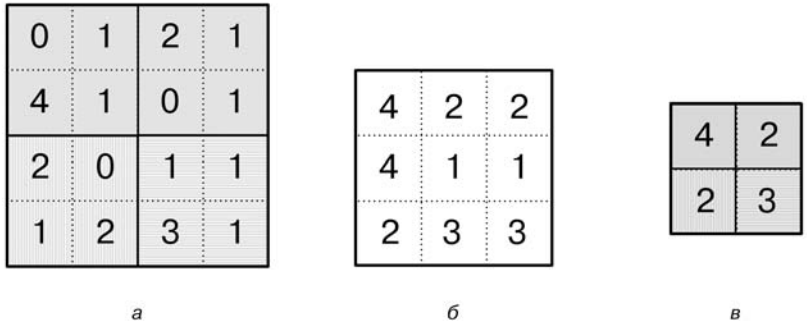


Рис. 5.3. Пример субдискретизации с окном размера 2 × 2: *а* — исходная матрица; *б* — матрица после субдискретизации с шагом 1; *в* — матрица после субдискретизации с шагом 2. Штриховка в исходной матрице *а* — соответствует окнам, по которым берется максимум с шагом 2; в части *в* — результат показан соответствующей штриховкой

Здесь d — это размер окна субдискретизации. Как правило, нас интересует случай, когда шаг субдискретизации и размер окна совпадают, то есть получаемая на вход матрица делится на непересекающиеся окна, в каждом из которых мы выбираем максимум; для $d = 2$ эта ситуация проиллюстрирована на рис. 5.3, *в*.

Хотя в результате субдискретизации действительно теряется часть информации, сеть становится более устойчивой к небольшим трансформациям изображения вроде сдвига или поворота.

Чтобы сделать все это в TensorFlow, как и в случае с операцией свертки, сначала зададим заглушку для входного «изображения» и приведем ее к нужной размерности:

```
x_inp = tf.placeholder(tf.float32, [4, 4])
x = tf.reshape(x_inp, [1, 4, 4, 1])
```

Теперь можно определять операции субдискретизации с помощью специальной функции `tf.nn.max_pool`; мы попробуем размеры шага 1 и 2:

```
x_valid = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 1, 1, 1],
                                                padding="VALID")
x_valid_half = tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
                                                padding="VALID")
```

Обратите внимание, что здесь появился не только аргумент `strides`, который мы уже обсуждали, но и отдельно задаваемый размер окна `ksize`. Он тоже представляет собой четырехмерный тензор, то есть субдискретизацию можно проводить по любым размерностям, включая элементы мини-батчей и каналы. Это может опять показаться несколько странным; например, что может означать субдискретизация

по каналам? Предположим, что перед нами стоит задача проверить, есть ли на данной фотографии попугай особого RGB-семейства, представители которого бывают только монохромными: идеального красного, зеленого или синего цвета. В таком случае мы можем решить, что есть смысл использовать субдискретизацию не только на пространственных измерениях, но и на каналах изображения, например для того, чтобы нейронная сеть могла как можно раньше избавиться от заведомо нерелевантных цветовых каналов. Звучит странно, но давайте попробуем теперь предположить, что на входе не отдельные фотографии, а видео в виде последовательных кадров. Тогда субдискретизация по каналам (если кадры соответствуют каналам) или мини-батчам (если входным примерам) внезапно становится очень осмысленной: соседние кадры почти всегда очень похожи, и если наша цель — распознать присутствие каких-то объектов в видео, то большую часть кадров можно спокойно выбросить.

Доведем наш эксперимент до конца. Для этого объявим входную матрицу, произведем вычисления и выпишем результат:

```
x = np.array([[0, 1, 2, 1],
              [4, 1, 0, 1],
              [2, 0, 1, 1],
              [1, 2, 3, 1]])

y_valid, y_valid_half = sess.run(
    [x_valid, x_valid_half],
    feed_dict={x_inp: x}
)

print "padding=VALID:\n", y_valid[0, :, :, 0]
print "padding=VALID, stride 2:\n", y_valid_half[0, :, :, 0]
```

Полученный выход опять ожидаем — в результате мы получили матрицы, изображенные на рис. 5.3:

```
padding=VALID:
[[ 4.  2.  2.]
 [ 4.  1.  1.]
 [ 2.  3.  3.]]
padding=VALID, stride 2:
[[ 4.  2.]
 [ 2.  3.]]
```

Итак, подведем промежуточный итог. Стандартный слой сверточной сети состоит из трех компонентов:

- свертка в виде линейного отображения, выделяющая локальные признаки;
- нелинейная функция, примененная покомпонентно к результатам свертки;
- субдискретизация, которая обычно сокращает геометрический размер получающихся тензоров.

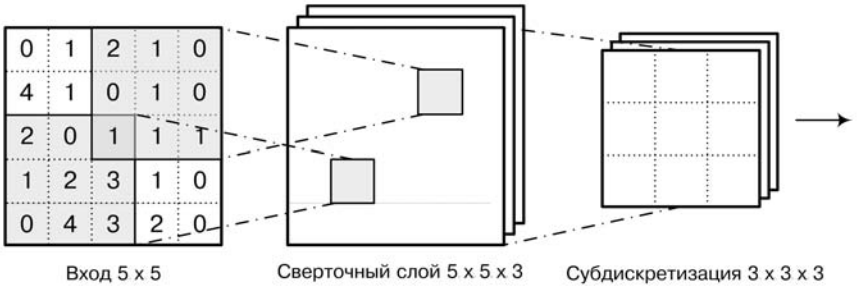


Рис. 5.4. Схема одного слоя сверточной сети: свертка, за которой следует субдискретизация

На рис. 5.4 мы изобразили такой слой в виде, в котором он представлен на стандартных изображениях сверточных сетей в статьях и руководствах. Отдельно рисовать нелинейность большого смысла нет, так что рисуют обычно две части: сначала свертку, потом субдискретизацию, указывая размерности. Обратите внимание, что по сравнению с «картинкой» на входе размерность тензора увеличилась: сверточная сеть обычно обучает сразу несколько карт признаков на каждом слое (на рис. 5.4 таких карт три).

Теперь о том, как обучить такие чудесные карты признаков. Как будут проходить градиенты в обратную сторону через сверточный слой, то есть как, собственно, мы будем обучать сверточную сеть? Предположим, что мы оптимизируем некоторую функцию ошибки E и уже знаем ее значения на выходах нашего сверточного слоя. Чтобы провести итерацию обучения, нужно понять, как через них выражаются значения градиентов функции ошибки от весов.

Через функцию взятия максимума ошибка проходит без изменений, слой субдискретизации ничего не обучает. Однако он делает проходящие по графу вычисления градиенты разреженными, ведь из всех элементов окна субдискретизации $z^l_{i,j}$ частная производная $\frac{\partial E}{\partial x^{l+1}_{i,j}}$ относится только к одному, максимальному, а остальные получат нулевой градиент, и на этом их обучение (на данном тренировочном примере!) можно будет считать законченным.

Пропускать через нелинейность мы тоже уже хорошо умеем: в обозначениях выше получится:

$$\frac{\partial E}{\partial y^l_{i,j}} = \frac{\partial E}{\partial z^l_{i,j}} \frac{\partial z^l_{i,j}}{\partial y^l_{i,j}} = \frac{\partial E}{\partial z^l_{i,j}} h'(y^l_{i,j}),$$

и здесь $\frac{\partial E}{\partial z^l_{i,j}}$ мы уже знаем, а $h'(y^l_{i,j})$ можем легко подсчитать.

А на сверточном уровне наконец-то появляются веса, которые нужно уметь обучать. Некоторая сложность здесь состоит в том, что все веса делятся, и каждый участвует во всех выходах; так что сумма получится достаточно большая:

$$\frac{\partial E}{\partial w^l_{a,b}} = \sum_i \sum_j \frac{\partial E}{\partial y^l_{i,j}} \frac{\partial y^l_{i,j}}{\partial w^l_{a,b}} = \sum_i \sum_j \frac{\partial E}{\partial z^{l-1}_{i+a,j+b}},$$

где индексы i и j пробегают все элементы картинки на промежуточном слое $y^l_{i,j}$, то есть после свертки, но до субдискретизации.

Для полноты картины осталось только пропустить градиенты на предыдущий слой. Это тоже несложно:

$$\frac{\partial E}{\partial x^l_{i,j}} = \sum_a \sum_b \frac{\partial E}{\partial y^l_{i-a,j-b}} \frac{\partial y^l_{i-a,j-b}}{\partial x^l_{i,j}} = \sum_i \sum_j \frac{\partial E}{\partial y^l_{i-a,j-b}} w_{a,b}.$$

Вот и все: мы адаптировали процедуру обратного распространения ошибки, она же обратный проход по графу вычислений, для сверточного слоя. Заметим еще, что обратный проход для свертки оказался очень похож на, опять же, свертку с теми же весами $w_{a,b}$, только вместо $i + a$ и $j + b$ теперь $i - a$ и $j - b$. В случае, когда изображение дополняется нулями по необходимости и размерности сохраняются, обратный проход можно считать в точности такой же сверткой, что и в прямом проходе, только с развернутыми осями.

А теперь можно построить глубокую сеть из таких слоев. Мы просто будем использовать выход очередного слоя как вход для следующего, а разные карты признаков будут служить каналами. Размер слоя за счет субдискретизации будет постепенно сокращаться, и в конце концов последние слои сети смогут «окинуть взглядом» весь вход, а не только маленькое окошечко из него. Именно такая архитектура была применена в знаменитой сети LeNet-5 [188], которая в конце 1990-х годов показала блестящие результаты на датасете MNIST для распознавания рукописных цифр[1]. После двух конструкций из свертки и субдискретизации в сети LeNet-5 следовали три полносвязных слоя с последовательно уменьшающимся числом нейронов, которые совмещали выделенные сверточными слоями локальные признаки и выдавали собственно ответ.

Прежде чем переходить к практике, последнее замечание об окнах сверточных слоев. В литературе эти окна называются *фильтрами*. Минимальный размер, при котором фильтр имеет центр и выражает такие отношения, как «слева»/«справа»

[1] Для своего времени, конечно. Но надо сказать, что начиная с LeNet и ее более поздних вариантов, набор данных MNIST понемногу потерял свое значение для переднего края моделей распознавания образов. В какой-то момент точность превысила 99 %, затем осталось около 40 ошибок из 10 000 примеров в валидационном множестве MNIST, и речь шла буквально о том, чтобы улучшить распознавание на данных конкретных примерах. Конечно, это уже не имеет большого практического значения, и современные модели для анализа изображений сравниваются на совсем других датасетах.

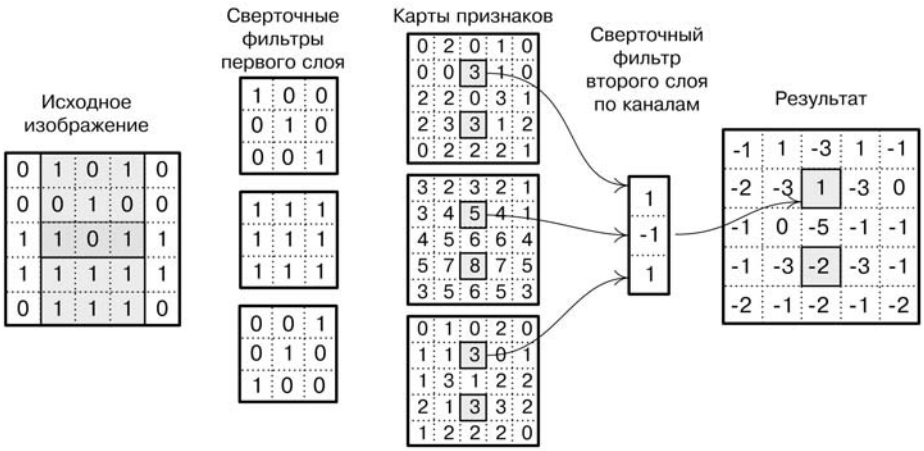


Рис. 5.5. Пример выделения признаков на двух сверточных слоях

или «сверху»/«снизу», — это, очевидно, размер 3×3. Именно такой размер фильтров используется в большинстве современных сверточных архитектур. К причинам этого мы вернемся чуть позже, а сейчас отметим чуть более экзотический вид сверток: свертки с фильтрами... 1×1.

Давайте разберемся, какой в этом смысл. Обычно, когда мы говорим о фильтрах, мы упоминаем только две размерности, отвечающие за «рецептивное поле» фильтра. Но на самом деле фильтр задается четырехмерным тензором, последние две размерности которого обозначают число каналов предшествующего и текущего слоя. Так, например, при работе с цветными изображениями на вход мы получаем три слоя, передающие соответственно красный, зеленый и синий цвета. Когда мы говорим, что в первом сверточном слое «фильтр размера 5×5», это значит, что в первом слое есть несколько (столько, сколько каналов мы хотим подать на вход в следующем слое) наборов весов, переводящих тензор размером $5 \times 5 \times 3$ («параллелепипед весов») в скаляр. Если это хорошенько осмыслить, станет ясно, откуда берутся фильтры размером 1×1: это просто линейные преобразования из входных каналов в выходные с последующей нелинейностью.

Теперь, познакомившись со всеми видами сверток, мы можем привести полноценный пример, который упрощенно, но вполне адекватно отражает то, что происходит при реальной обработке изображений сверточными сетями. Пример проиллюстрирован на рис. 5.5:

- каждая карта признаков первого слоя выделяет некий признак в каждом окне исходного изображения; в частности, первая карта признаков «ищет» диагональную линию из единичек, точнее, из пикселов высокой интенсивности (это значит, что она сильнее всего активируется, когда пикселы на диагонали

присутствуют), вторая просто активируется на закрашенное окно изображения (чем больше пикселов с высокой интенсивностью, тем лучше), а третья аналогична первой и ищет другую диагональ;
- нелинейность в сверточном слое и слой субдискретизации мы в этом примере решили пропустить;
- карта признаков на втором слое (для простоты одна) пытается найти на картинке крестик из двух диагональных линий; для этого она объединяет признаки, выделенные на первом слое, то есть свертка второго слоя — это одномерная свертка по каналам (признакам), а не по окнам в изображении; вот что свертка второго слоя «хочет увидеть» в том или ином наборе признаков:
  - как можно более ярко выраженную диагональную линию из левого верхнего в правый нижний угол, то есть сильно активированный первый признак первого слоя;
  - ярко выраженную линию из левого нижнего в правый верхний угол, то есть сильно активированный третий признак первого слов;
  - и как можно меньше других активированных пикселов, то есть суммарная активация, выраженная во втором признаке первого слоя, играет для этой свертки в минус;
- в результате такая сеть действительно находит крестики на исходной картинке; обратите внимание на разницу между двумя выделенными на рис. 5.5 окнами — в одном действительно получается крестик, а в другом кроме крестика есть еще много «мусора», и за счет второго признака нейрон, отвечающий за крестик, получает отрицательную активацию;
- а если бы субдискретизация была нетривиальной, она находила бы ещё и «псевдокрестики», в которых диагональные линии находятся рядом друг с другом, а не обязательно в одном и том же окне.

Конечно, свертка сквозь все каналы предшествующего слоя не является единственным возможным вариантом. В TensorFlow реализованы несколько альтернативных видов свертки. Мы не будем подробно их разбирать в этой главе, так как уверены, что интересующийся читатель сможет самостоятельно разобраться с тонкостями, однако один пример приведем. Функция `depthwise_conv2d` — свертка «по глубине» — задается тензором следующей размерности:

```
[filter_height, filter_width, in_channels, channel_multiplier]
```

И каждый входной канал по отдельности «сворачивается» в `channel_multiplier` выходных каналов. В этом случае как раз не стоит забывать, что мы не просто преобразуем входные каналы в выходные, а делаем пространственную свертку.

Мы поговорим о современных архитектурах сверточных сетей, в которых иногда используются и такие экзотические свертки, в разделе 5.4. Но сначала — практические упражнения!

5.3. Пример: свертки для распознавания цифр

> Между лопатками великого комбинатора лиловели и переливались нефтяной радугой синяки странных очертаний.
>
> — Честное слово, цифра восемь! — воскликнул Воробьянинов. — Первый раз вижу такой синяк.
>
> — А другой цифры нет? — спокойно спросил Остап.
>
> *И. Ильф, Е. Петров. 12 стульев*

Теперь, когда теоретическая часть сверточных сетей более или менее ясна и мы увидели как абстрактную мотивацию, так и конкретные примеры сверточных архитектур, давайте попробуем применить все это на практике. В этом разделе мы вернемся к распознаванию рукописных цифр из датасета MNIST и попробуем существенно улучшить наше решение из раздела 3.6. Начнем снова с загрузки данных и импорта библиотек:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Этот код скачает датасет MNIST (если вы его еще не импортировали раньше), а затем преобразует правильные ответы из него в one-hot представление: правильными ответами станут векторы размерности десять, в которых одна единица на месте нужной цифры.

Зададим заглушки для тренировочных данных:

```
x = tf.placeholder(tf.float32, [None, 784])
y = tf.placeholder(tf.float32, [None, 10])
```

В разделе 3.6 мы считали вход просто вектором длины 784. Это, конечно, сильно осложняло задачу нейронной сети: значения разных пикселов получались совершенно независимыми друг от друга, и мы полностью теряли информацию о том, какие из них расположены ближе друг к другу и, соответственно, должны больше влиять друг на друга. На этот раз мы будем применять сверточные сети, для которых пространственная структура изображений важна и в которых она постоянно используется. Поэтому переформатируем входной вектор в виде двумерного массива из 28×28 пикселов:

```
x_image = tf.reshape(x, [-1,28,28,1])
```

Обратите внимание, что, как мы уже объясняли выше, формально массив теперь четырехмерный: первая размерность -1 соответствует заранее неизвестному размеру мини-батча, а в четвертой размерности мы указали, что в каждом пикселе стоит только одно число. Для цветной картинки, например, в каждом пикселе могли бы стоять три числа, соответствующие интенсивностям красного, зеленого и синего цветов (RGB).

Дальше нужно создать сверточный слой. Во-первых, мы должны выбрать размер ядра свертки — для этого примера давайте возьмем ядро размера 5×5. Во-вторых, нам нужно определиться с числом фильтров, которые мы будем обучать; пусть на первом слое их будет 32. И, в-третьих, мы уже разобрались с числом каналов в нашем изображении, то есть с тем, сколько чисел задают каждый пиксел: так как датасет черно-белый, цветовой канал всего один. Итак, можно создавать переменные для весов свертки:

```
W_conv_1 = tf.Variable(tf.truncated_normal([5, 5, 1, 32], stddev=0.1))
```

Давайте тщательно разберем эту строчку. Функцию `tf.truncated_normal` мы уже видели: мы инициализируем веса с помощью обрезанного нормального распределения с заданным стандартным отклонением 0,1 и ожиданием 0. Такое распределение выбрано потому, что мы планируем в качестве функции активации использовать ReLU. Массив из четырех чисел на входе — это форма тензора, который мы инициализируем с помощью нормального распределения. Первые два параметра задают размер ядра, третий отвечает за число входных каналов, а четвертый определяет собственно число выходных каналов. По сути, двигая наше окно-фильтр по исходному изображению, мы на выходе получаем вместо одного значения столбик из 32 значений, что можно представить себе как применение 32 разных фильтров.

Итак, с весами разобрались, осталось только задать свободные члены (biases); несмотря на сложную структуру тензора весов, для свободных членов достаточно отвести всего 32 переменные: для каждого из 32 фильтров, независимо от того, к какой именно области изображения он приложен, результат свертки сдвигается на одно и то же число.

```
b_conv_1 = tf.Variable(tf.constant(0.1, shape=[32]))
```

Обратите внимание на то, как мало у нас здесь переменных: весов на сверточном слое всего $5 \cdot 5 \cdot 1 \cdot 32 = 800$, да еще 32 свободных члена. Когда мы в разделе 3.6 задавали полносвязный слой для работы с теми же MNIST-изображениями, весов получалось $784 \cdot 10 = 7840$; а если бы мы захотели добавить скрытый слой размером 32, как здесь, то на первом слое весов стало бы $784 \cdot 32 = 25\,088$, что гораздо больше. Это яркая иллюстрация того, как сверточные сети используют дополнительную информацию о структуре входов для того, чтобы делать такую «абсолютную регуляризацию», объединяя массу весов. Мы знаем, что изображение имеет двумерную структуру, знаем его геометрию и заранее определяем, что хотели бы обрабатывать каждое окно в изображении одними и теми же фильтрами: нам все равно, в какой части картинки будут расположены штрихи, определяющие цифру 5, нужно просто распознать, что это именно 5, а не 8.

Теперь у нас определены переменные для всех весов сверточного слоя; что с ними делать дальше, TensorFlow знает сам:

```
conv_1 = tf.nn.conv2d(
    x_image, W_conv_1, strides=[1, 1, 1, 1], padding="SAME") + b_conv_1
```

Сама функция `tf.nn.conv2d` только применяет сверточные фильтры, она делает линейную часть работы, а функцию активации нам нужно задать самостоятельно потом. В качестве функции активации, как и собирались, возьмем ReLU:

```
h_conv_1 = tf.nn.relu(conv_1)
```

Итак, слой фильтров с нелинейностью готов. Чтобы соблюсти стандартную архитектуру сверточной сети, осталось только добавить слой субдискретизации:

```
h_pool_1 = tf.nn.max_pool(
    h_conv_1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")
```

Функция `tf.nn.max_pool` определяет max-pooling слой, выбирая максимальное значение из каждого окна. Параметр `ksize` здесь как раз и задает размер этого окна, в котором мы выбираем максимальный элемент. Он имеет ту же структуру, что и `strides`. Обратите внимание, что здесь уже вполне можно представить себе ситуацию, когда первая компонента будет не равна единице и мы захотим выбирать «самые подходящие» из нескольких последовательных изображений. Можно даже задать первую размерность -1, тогда слой субдискретизации будет выбирать «самое подходящее» изображение из всего мини-батча.

Параметры `strides` и `padding` обозначают здесь то же самое, что и для сверточного слоя, только в этот раз мы двигаемся по изображению в обе стороны с шагом 2. Понятно, что после этого слоя размер изображения в обоих направлениях уменьшится вдвое, до 14×14.

Раз весов у нас не так уж и много, давайте по той же схеме, что и выше, добавим еще один сверточный слой и слой субдискретизации, в этот раз используя на этом слое 64 фильтра:

```
W_conv_2 = tf.Variable(tf.truncated_normal([5, 5, 32, 64], stddev=0.1))
b_conv_2 = tf.Variable(tf.constant(0.1, shape=[64]))

conv_2 = tf.nn.conv2d(
    h_pool_1, W_conv_2, strides=[1, 1, 1, 1], padding="SAME") + b_conv_2

h_conv_2 = tf.nn.relu(conv_2)
h_pool_2 = tf.nn.max_pool(
    h_conv_2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding="SAME")
```

Как правило, в глубоких нейронных сетях за сверточными слоями следуют полносвязные, задача которых состоит в том, чтобы «собрать вместе» все признаки из фильтров и собственно перевести их в самый последний слой, который выдаст ответ. Но для начала нам нужно из двумерного слоя сделать плоский; в TensorFlow это делается функцией `reshape`:

```
h_pool_2_flat = tf.reshape(h_pool_2, [-1, 7*7*64])
```

Число $7 \cdot 7 \cdot 64$ возникло из-за того, что мы дважды применили субдискретизацию и при этом в последнем слое использовали 64 фильтра. И теперь осталось

только добавить полносвязные слои. В этот раз мы не будем подробно расписывать добавление полносвязных слоев, так как это мы уже не раз делали раньше. Добавляем первый слой из 1024 нейронов:

```
W_fc_1 = tf.Variable(tf.truncated_normal([7*7*64, 1024], stddev=0.1))
b_fc_1 = tf.Variable(tf.constant(0.1, shape=[1024]))
h_fc_1 = tf.nn.relu(tf.matmul(h_pool_2_flat, W_fc_1) + b_fc_1)
```

Регуляризуем его дропаутом:

```
keep_probability = tf.placeholder(tf.float32)
h_fc_1_drop = tf.nn.dropout(h_fc_1, keep_probability)
```

Теперь добавляем второй, самый последний слой с десятью выходами:

```
W_fc_2 = tf.Variable(tf.truncated_normal([1024, 10], stddev=0.1))
b_fc_2 = tf.Variable(tf.constant(0.1, shape=[10]))

logit_conv = tf.matmul(h_fc_1_drop, W_fc_2) + b_fc_2
y_conv = tf.nn.softmax(logit_conv)
```

Осталось определить ошибку и ввести оптимизатор:

```
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(logit_conv, y))
train_step = tf.train.AdamOptimizer(0.0001).minimize(cross_entropy)
```

В этот раз мы используем алгоритм оптимизации Adam, о котором уже говорили в разделе 4.5. Оцениваем точность:

```
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

И осталось только запустить обучение и дождаться результата:

```
init = tf.initialize_global_variables()
sess = tf.Session()
sess.run(init)
for i in range(10000):
    batch_xs, batch_ys = mnist.train.next_batch(64)
    sess.run(train_step,
        feed_dict={x: batch_xs, y: batch_ys, keep_probability: 0.5})

print(sess.run(accuracy,
    feed_dict={x: mnist.test.images, y: mnist.test.labels, keep_probability: 1.}))
>> 0.9906
```

А теперь давайте для сравнения сделаем то же самое, но в Keras. Суть процесса, конечно, не меняется, но кода получается заметно меньше. Как и в TensorFlow, в Keras набор данных MNIST можно загрузить средствами самой библиотеки:

```
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.utils import np_utils

(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Затем давайте подготовим данные к тому, чтобы подать их на вход сети. Здесь нам потребуются три вспомогательные процедуры. Во-первых, нужно будет, как и раньше, превратить каждую картинку в двумерный массив:

```
batch_size, img_rows, img_cols = 64, 28, 28
X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
input_shape = (img_rows, img_cols, 1)
```

Во-вторых, входные данные MNIST в Keras, в отличие от TensorFlow, представляют собой целые числа от 0 до 255; можно было бы обучать сеть и на таких данных, но давайте для единообразия все-таки приведем их к типу `float32` и нормализуем от 0 до 1, как раньше:

```
X_train = X_train.astype("float32")
X_test = X_test.astype("float32")
X_train /= 255
X_test /= 255
```

В-третьих, переведем правильные ответы в one-hot представление; для этого служит вспомогательная процедура `to_categorical` из `keras.np_utils`:

```
Y_train = np_utils.to_categorical(y_train, 10)
Y_test = np_utils.to_categorical(y_test, 10)
```

Теперь пора задавать собственно модель. В Keras это выглядит примерно так же, как в TensorFlow, но некоторые названия и параметры более узнаваемы. Сначала инициализируем модель:

```
model = Sequential()
```

Теперь добавим сверточные слои. Нам понадобится слой `Convolution2D`, основными аргументами которого являются число фильтров и размер окна, аргумент `border_mode` имеет ровно тот же смысл, что аргумент `padding` в TensorFlow, а аргумент `input_shape` сообщает Keras, какой размерности тензор ожидать на входе:

```
model.add(Convolution2D(32, 5, 5, border_mode="same", input_shape=input_shape))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), border_mode="same"))
model.add(Convolution2D(64, 5, 5, border_mode="same", input_shape=input_shape))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2), border_mode="same"))
```

В этом примере мы сохранили ту же архитектуру, что была выше. Обратите внимание, что в Keras нам не нужно отдельно инициализировать переменные, которые будут затем использоваться в качестве весов или свободных членов в слоях: Keras сам понимает, сколько должно быть весов у той или иной сверточной архитектуры, и сам поймет, что по ним нужно модель оптимизировать. Единственное, чем ему нужно для этого помочь, — задать явно размерность `input_shape`.

Полносвязные слои тоже получаются достаточно просто. За «переформатирование» тензора, которое в TensorFlow делалось функцией `tf.reshape`, теперь отвечает дополнительный «слой», который называется `Flatten` и способен сам понять, тензор какой формы подается ему на вход:

```
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation("relu"))
model.add(Dropout(0.5))
model.add(Dense(10))
model.add(Activation("softmax"))
```

И все, можно компилировать и запускать обучение:

```
model.compile(loss="categorical_crossentropy",
              optimizer="adam", metrics=["accuracy"])
model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=10,
          verbose=1, validation_data=(X_test, Y_test))
score = model.evaluate(X_test, Y_test, verbose=0)
print("Test score: %f" % score[0])
print("Test accuracy: %f" % score[1])
```

На выходе получится примерно такая картина (мы сокращаем вывод, чтобы строки помещались в ширину страницы):

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 - loss: 0.1262 - acc: 0.9621 - val_loss: 0.0514 - val_acc: 0.9830
Epoch 2/10
60000/60000 - loss: 0.0436 - acc: 0.9864 - val_loss: 0.0320 - val_acc: 0.9892
Epoch 3/10
60000/60000 - loss: 0.0296 - acc: 0.9909 - val_loss: 0.0268 - val_acc: 0.9922
Epoch 4/10
60000/60000 - loss: 0.0242 - acc: 0.9925 - val_loss: 0.0313 - val_acc: 0.9898
Epoch 5/10
60000/60000 - loss: 0.0170 - acc: 0.9944 - val_loss: 0.0239 - val_acc: 0.9928
Epoch 6/10
60000/60000 - loss: 0.0164 - acc: 0.9950 - val_loss: 0.0205 - val_acc: 0.9936
Epoch 7/10
60000/60000 - loss: 0.0131 - acc: 0.9959 - val_loss: 0.0290 - val_acc: 0.9922
```

```
Epoch 8/10
60000/60000 - loss: 0.0117 - acc: 0.9963 - val_loss: 0.0259 - val_acc: 0.9920
Epoch 9/10
60000/60000 - loss: 0.0103 - acc: 0.9970 - val_loss: 0.0331 - val_acc: 0.9903
Epoch 10/10
60000/60000 - loss: 0.0096 - acc: 0.9973 - val_loss: 0.0321 - val_acc: 0.9915
Test score: 0.03208116913667295
Test accuracy: 0.99150000000000005
```

Как видите, такая же сеть в Keras и работает примерно так же, достигая в данном случае точности в 99,15 % на валидационном множестве MNIST.

Кстати, этот пример позволяет нам упомянуть еще одну интересную возможность библиотеки Keras. Обратите внимание, что после шестой эпохи результат получался даже лучше: 99,36 % на валидационном множестве. Не исключено, что после этого у модели начался легкий оверфиттинг[1], и нам хотелось бы в итоге использовать веса модели после шестой эпохи, а не после десятой; в разделе 4.1 мы уже обсуждали этот бесхитростный метод «регуляризации» — *раннюю остановку* (early stopping).

Можно, конечно, написать цикл по эпохам обучения, проверять в этом цикле ошибку на валидационном множестве, сохранять веса модели в файл, а затем выбрать файл, соответствующий самой лучшей эпохе. Но оказывается, что в Keras все это можно выразить буквально в аргументах функции `fit`! Вот как обычно выглядит запуск обучения сложных моделей на практике:

```
model.fit(X_train, Y_train,
  callbacks=[ ModelCheckpoint("model.hdf5", monitor="val_acc",
    save_best_only=True, save_weights_only=False, mode="auto")],
  validation_split=0.1, nb_epoch=10, batch_size=64)
```

Здесь аргумент `callbacks` позволяет задать функции, запускаемые после каждой эпохи обучения. Можно написать эти функции самому, а можно воспользоваться стандартными. Так, функция `ModelCheckpoint` из модуля `keras.callbacks` — это вспомогательная процедура, которая после каждой эпохи обучения сохраняет модель в файл, имя которого подается на вход, в данном случае `model.hdf5`. А параметр `save_best_only=True` позволяет после каждой эпохи проверять метрику качества, заданную в параметре `monitor` (в данном случае `val_acc`, то есть точность на валидационном множестве), и перезаписывать сохраняемую модель только в том случае, если эта метрика улучшилась. Обратите также внимание на параметр `validation_split`: если у вас нет заранее выделенного тренировочного и тестового

[1] Для этой модели и датасета MNIST оверфиттинг вряд ли очень уж вероятен, но в целом на практике часто бывает так, что лучшие значения метрик качества на валидационном множестве получаются где-то в начале процесса обучения, а потом, хотя ошибка на тренировочном множестве продолжает падать, ошибка на валидационном множестве только растет. В таких случаях и может пригодиться то, о чем мы сейчас говорим.

подмножеств (у нас они были сразу в датасете MNIST), то можно просто попросить Keras использовать для валидации случайное подмножество входных данных, составляющее заданную их долю, в данном случае 10 %.

5.4. Современные сверточные архитектуры

> Когда читаешь газеты и журналы за последнее время, то иногда приходится с недоумением взглянуть на дату издания: не попался ли случайно в руки листок, писанный тому два, три века назад?
>
> *П. Л. Лавров. Хаос буржуазной цивилизации за последнее время*

В этом разделе мы поговорим о том, как развиваются современные сверточные архитектуры и куда, вообще говоря, движется сейчас анализ изображений. Сначала давайте вернемся к глубоким архитектурам и, в частности, к сверткам с фильтрами размера 3×3. Одной из самых популярных глубоких сверточных архитектур является модель, которую принято называть VGG [507]. Название происходит от того, что эта модель была разработана в Оксфордском университете в Группе визуальной геометрии (Visual Geometry Group), и их модели, представленные на ряд конкурсов по компьютерному зрению, выступали там под кодовым названием VGG. Соревнования проходили в 2014 году, что делает VGG самой «старой» из моделей, представленных в этом разделе.

VGG — это на самом деле сразу две конфигурации сверточных сетей, на 16 и 19 слоев. Основным нововведением, из-за которого мы и рассказываем о VGG, стала идея использовать фильтры размером 3×3 с единичным шагом свертки вместо использовавшихся в лучших моделях предыдущих лет сверток с фильтрами 7×7 с шагом 2 [407, 585] и 11×11 с шагом 4 [284]. Причем это не просто утверждение из разряда «мы попробовали, и стало лучше», а хорошо аргументированное предложение; давайте попробуем разобраться в аргументах.

Во-первых, рецептивное поле трех подряд идущих сверточных слоев размером 3×3 имеет размер 7×7, в то время как весов у них будет всего 27, против 49 в фильтре 7×7. Аналогично обстоит дело и с фильтрами 11×11. Это значит, что VGG может стать более глубокой, то есть содержать больше слоев, при этом одновременно *уменьшая* общее число весов. Конечно, для того чтобы это было правдой, между соответствующими сверточными слоями не должно быть слоев субдискретизации.

Во-вторых, наличие дополнительной нелинейности между слоями позволяет увеличить «разрешающую способность» по сравнению с единственным слоем с большей сверткой. Этот же аргумент можно использовать как мотивацию для того, чтобы ввести в сеть свертки размером 1×1; такие слои тоже позволяют добавить дополнительную нелинейность в сеть, не меняя размер рецептивного поля.

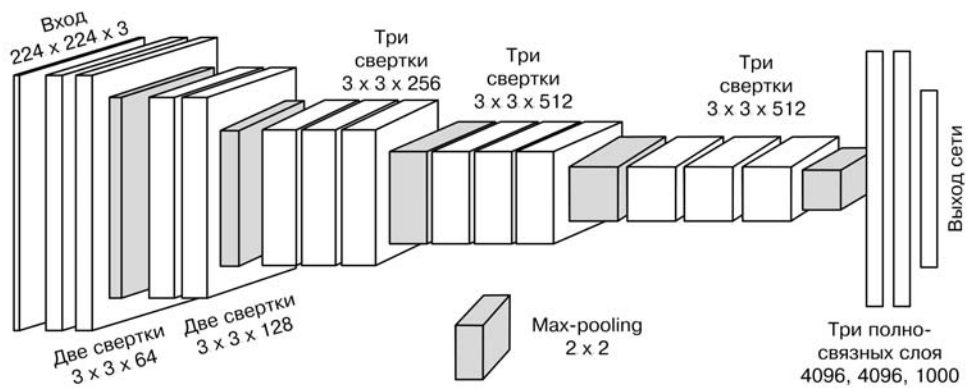


Рис. 5.6. Схема сети VGG-16

Полученная в итоге модель в 2014 году одержала победу в одной из номинаций известного соревнования по компьютерному зрению, ImageNet Large Scale Visual Recognition Competition (ILSVRC) [244]. А популяризация сверточных слоев 3×3 привела, в частности, к тому, что NVIDIA в очередном релизе библиотеки cuDNN специально оптимизировала работу с такими свертками.

Схема одной из VGG-сетей показана на рис. 5.6. Обратите внимание на три вещи: во-первых, как по две-три свертки 3×3 следуют друг за другом без субдискретизации; во-вторых, как число карт признаков постепенно растет на более глубоких уровнях сети; в-третьих, как в конце полученные признаки окончательно «сплющиваются» в одномерный вектор и на нем работают последние, уже полносвязные слои. Все это стандартные методы создания архитектур глубоких сверточных сетей, и если вы будете разрабатывать свою архитектуру, вам наверняка стоит следовать этим общим принципам.

Кстати, веса уже готовых моделей, обученных на больших наборах данных, например ImageNet, можно найти в Интернете. В частности, с сайта авторов модели можно скачать веса и конфигурации моделей для популярной библиотеки для работы с нейронными сетями Caffe. Это общее место для многих современных моделей компьютерного зрения: дело в том, что даже с современными мощностями датасеты настолько большие, а обучение настолько долгое и сложное (например, в [507] сказано, что каждый вариант VGG в 2014 году обучали две-три недели на четырех лучших на тот момент видеокартах), что проще скачать готовые веса и, возможно, немного дообучить их для вашей конкретной задачи.

Следующая важная сверточная архитектура — это архитектура Inception [181]. Она была разработана в Google и появилась практически одновременно с VGG, в сентябре 2014 года; команда GoogLeNet победила с этой сетью в нескольких номинациях все того же конкурса ILSVRC-2014. Авторы архитектуры вдохновились

идеями из работы 2012 года *Network In Network* [328], в которой была предложена идея использовать в качестве строительных блоков для глубоких сверточных сетей не просто последовательность «свертка — нелинейность — субдискретизация», как это обычно делается, а более сложные конструкции.

В [328] такими конструкциями были полноценные (но маленькие) нейронные сети с полносвязными слоями. А в Inception такой компонент «собирают» из небольших сверточных конструкций. Так что название сети отражает не только «глубину» из одноименного фильма, но и развитую там идею «вложенной» архитектуры: сон внутри сна внутри сна...

Эта работа содержит несколько крайне интересных и важных идей, благодаря которым, в частности, несмотря на большую заявленную глубину — 22 слоя без учета субдискретизации — у Inception на самом деле меньше параметров, чем у VGG!

Но давайте по порядку. «Строительными блоками» Inception являются модули, комбинирующие свертки размером 1×1, 3×3 и 5×5, а также max-pooling субдискретизацию. Каждый блок представляет собой объединение четырех «маленьких» сетей, выходы которых объединяются в выходные каналы и передаются на следующий слой. Выбор набора сверток и субдискретизации обусловлен скорее удобством, чем необходимостью, и при желании читатель может поэкспериментировать с другими конфигурациями. Команда GoogLeNet, разработавшая Inception, во второй версии своей модели тоже пересмотрела архитектуру этих модулей.

Одно из ключевых нововведений — использование сверточных слоев 1×1 не столько в качестве дополнительной нелинейности, сколько для понижения размерности между слоями. Свертки 3×3 и тем более 5×5 между слоями с большим числом каналов (а в Inception-модулях каналов может быть вплоть до 1024), оказываются крайне ресурсоемкими, несмотря на малые размеры отдельно взятых фильтров. А фильтры 1×1 могут помочь сократить число каналов, прежде чем подавать их на фильтры большего размера.

Эта идея отражена на рис. 5.7, *а*, на котором показана структура одного блока из исходной работы [181]. А на рис. 5.7, *б* представлена очень общая и высокоуровневая схема всей сети: она начинается с двух «обычных» сверточных слоев, а затем идут 11 Inception-модулей, дважды перемежаемых субдискретизацией, которая понижает размерность; после этого сеть завершается традиционными полносвязными слоями, дающими уже собственно выход классификатора.

Помимо конфигурации Inception-модулей и понижения размерности с помощью сверток 1×1, в работе [181] представлена еще одна важная идея. С учетом достаточно глубокой архитектуры сети — а в общей сложности *GoogLeNet* содержит порядка 100 различных слоев с общей глубиной в 22 параметризованных слоя, или 27 слоев с учетом субдискретизаций — эффективное распространение градиентов по ней вызывает сомнения. Чтобы решить эту проблему, авторы предложили добавить вспомогательные классифицирующие сети поверх некоторых промежуточных слоев.

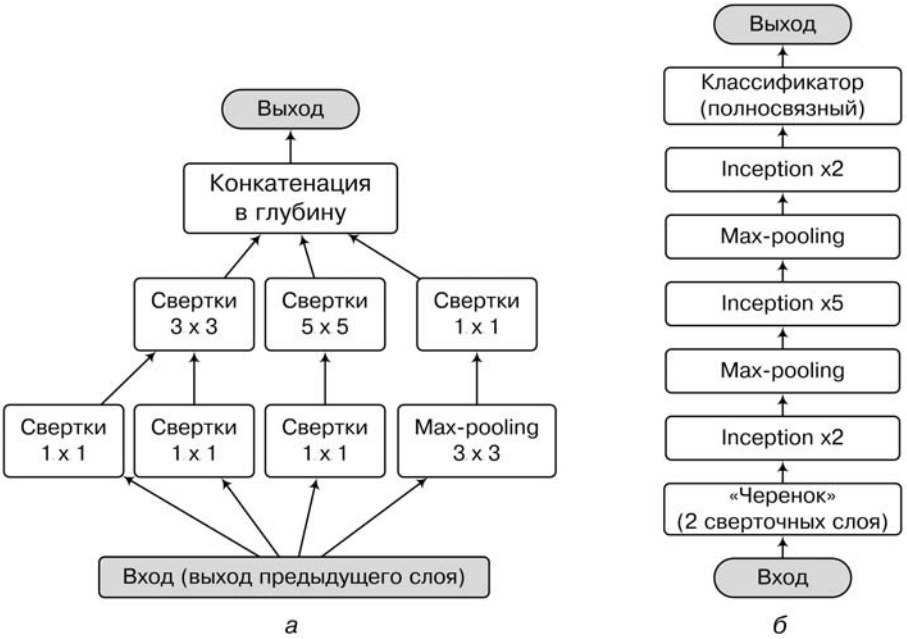


Рис. 5.7. Inception: *а* — схема одного Inception-модуля; *б* — общая схема сети GoogLeNet

Иначе говоря, мы добавляем две новые небольшие полносвязные сети, делающие предсказания на основе промежуточных признаков, выводим из них ту же функцию ошибки классификации и обучаем на той же задаче не только всю сеть, но и отдельно первую ее часть, а также первую и вторую. В архитектуре [181] присутствуют две такие дополнительные сети, состоящие из субдискретизации усреднением, свертки 1×1, полносвязного слоя, дропаута и линейного слоя с softmax в качестве функции ошибки классификатора. При обучении модели ошибка от этих подсетей добавляется к общей функции ошибки с понижающим коэффициентом 0,3. Потенциально этот трюк должен был ускорить обучение нижних слоев на ранних этапах обучения и привести к более быстрой сходимости, а в итоге оказалось, что сходимость не сильно ускорилась, но зато улучшилось окончательное решение.

Кстати, о следующей версии. Через год с небольшим практически та же команда авторов опубликовала следующую работу [448]. В ней в общую структуру Inception-модулей внесли новое важное изменение: заменили практически все «большие» свертки на композиции сверток размерности 3×3 и $1 \times n$. Выше мы уже обсуждали в контексте VGG, что свертку размера 5×5 можно заменить двумя последовательными сверточными слоями, каждый размера 3×3, при этом не потеряв в выразительной силе и сократив общее число весов. Однако авторы второй версии *Inception* пошли дальше и предложили заменить свертки произвольного размера

$n \times n$ на два последовательных слоя размером $n \times 1$ и $1 \times n$. Идейно (но не формально!) это соответствует сингулярному разложению матрицы весов свертки, то есть разложению в произведение двух прямоугольных матриц. Такой подход существенно сокращает вычислительную сложность модели даже по сравнению с факторизацией на свертки 3×3. Экспериментируя с такими слоями, авторы пришли к выводу, что в слоях, где размер карты признаков находится в пределах от 12×12 до 20×20 нейронов, декомпозиция в пары сверток 7×1 и 1×7 показывает хорошие результаты. Кроме того, свертка 5×5 из базовой конфигурации *Inception*-модуля была заменена на две последовательные свертки 3×3.

Были новости и о вспомогательных классификаторах. Новые эксперименты показали, что на ранних эпохах обучения эффект от них по-прежнему не заметен, однако ближе к концу обучения модель с дополнительным классификатором обучается лучше, попадает в более хороший локальный оптимум. В связи с этим представляется, что вспомогательные сети не способствуют обучению низкоуровневых признаков как таковому, а скорее служат в качестве регуляризатора сети. Кроме того, во второй версии *Inception* выяснилось, что дополнительные слои нормализации по мини-батчам или дропаута во вспомогательной сети приводят к дальнейшему улучшению общего решения.

Можно сказать, что с сети VGG началась эпоха «по-настоящему глубокого» обучения. Наконец-то специалисты по машинному обучению смогли эффективно обучать модели глубже, чем «несколько слоев», и показывать результаты, сравнимые или существенно лучшие, чем на тех же датасетах способен показать человек. Так, например, задачу классификации изображений на датасете CIFAR-10 современные нейронные сети решают с точностью около 96.5 %, в то время как точность человеческих результатов составляет около 94 % [37, 272]. Однако для того, чтобы обучать еще более глубокие нейронные сети, потребовалось не только собрать воедино все идеи из главы 4 и этой главы, но и добавить еще один важный трюк.

После того как группа Хинтона нашла способ предобучать слой за слоем нейронные сети любой глубины, Ян ЛеКун [135] и Йошуа Бенджи совместно с Хавьером Глоро [179] предложили эффективные методы инициализации весов, а Иоффе и Сегеди добавили промежуточные слои, нормализующие выходы по минибатчам [252], проблема затухающих градиентов, которая долгое время преследовала нейронные сети, наконец-то отошла на второй план.

Как показала практика, глубокие архитектуры стало возможно обучать эффективно, однако те решения, к которым сходились нейронные сети большой глубины, часто оказывались хуже, чем у менее глубоких моделей. И эта «деградация» не была связана с переобучением, как можно было бы предположить. Оказалось, что это более фундаментальная проблема: с добавлением новых слоев ошибка растет не только на тестовом, но и на тренировочном множестве. Хотя, казалось бы, более «мелкая» сеть — это частный случай более глубокой: мы ведь могли бы обучить менее глубокую сеть, а затем ее веса использовать для инициализации более глубокой; дополнительные слои при этом можно просто инициализировать так, чтобы

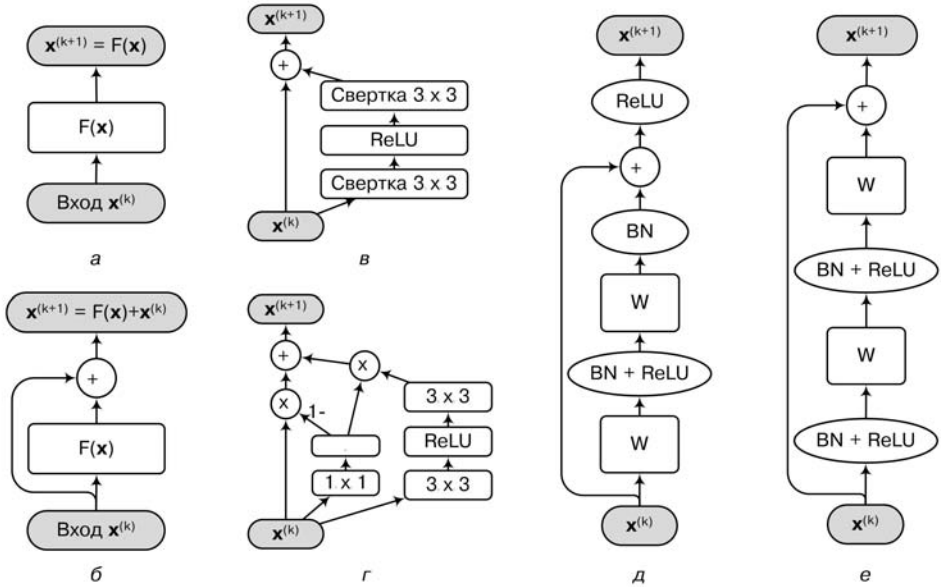


Рис. 5.8. Блоки сетей для остаточного обучения: *а* — базовый блок; *б* — такой же блок с остаточной связью; *в* — простой блок с остаточной связью; *г* — блок с остаточной связью, контролирующийся гейтом; *д* — блок с остаточной связью исходной сети ResNet [111]; *е* — более поздняя модификация [242]

они копировали вход в выход. Тогда ошибка более глубокой сети по определению не будет выше, чем ошибка ее подсети, а после обучения мы ожидаем улучшения. Но эксперименты показывают, что этого не происходит: более сложные модели сложнее обучать, и даже к такому тривиальному для нас решению современные методы обучения за разумное время не приводят.

Для решения проблемы деградации команда из Microsoft Research разработала новую идею: *глубокое остаточное обучение* (deep residual learning), которое легло в основу сети *ResNet* [111], а также многих последующих работ. В базовой структуре новой модели нет ничего нового: это слои, идущие последовательно друг за другом. Отдельные уровни, составные блоки сети тоже выглядят достаточно стандартно, это просто сверточные слои, обычно с дополнительной нормализацией по мини-батчам. Разница в том, что в остаточном блоке слой из нейронов можно «обойти»: есть специальная связь между выходом предыдущего слоя $\boldsymbol{x}^{(k)}$ и следующего слоя $\boldsymbol{x}^{(k+1)}$, которая идет напрямую, не через вычисляющий что-то слой.

Базовый слой нейронной сети на рис. 5.8, *а* превращается в остаточный блок с обходным путем на рис. 5.8, *б*. Математически происходит очень простая вещь: когда два пути, «сложный» и «обходной», сливаются обратно, их результаты просто складываются друг с другом. И остаточный блок выражает такую функцию:

$$\boldsymbol{y}^{(k)} = F(\boldsymbol{x}^{(k)}) + \boldsymbol{x}^{(k)},$$

где $\boldsymbol{x}^{(k)}$ — входной вектор слоя k, $F(x)$ — функция, которую вычисляет слой нейронов, а $\boldsymbol{y}^{(k)}$ — выход остаточного блока, который потом станет входом следующего слоя $\boldsymbol{x}^{(k+1)}$.

Получается, что если блок в целом должен апроксимировать функцию $H(\boldsymbol{x})$, то это достигается тогда, когда $F(\boldsymbol{x})$ аппроксимирует остаток (residue) $H(\boldsymbol{x}) - \boldsymbol{x}$, отсюда и название *остаточные сети* (residual networks). В остаточном блоке мы обучаем слой нейронов воспроизводить *изменения* входных значений, необходимые для получения итоговой функции.

Что в этом хорошего? Во-первых, часто получается, что обучить «остаточную» функцию проще, чем исходную; в [111] авторы приводят в пример тождественную функцию $h(\boldsymbol{x}) = \boldsymbol{x}$. Оказывается, что с помощью двухслойной нелинейной нейронной сети выучить тождественную функцию достаточно сложно; в то же время в остаточной форме от сети требуется просто заполнить все веса нулями, а «обходной путь» сделает всю работу сам.

Главная же причина состоит в том, что градиент во время обратного распространения может проходить через этот блок беспрепятственно, градиенты не будут затухать, ведь всегда есть возможность пропустить градиент напрямую:

$$\frac{\partial \boldsymbol{y}^{(k)}}{\partial \boldsymbol{x}^{(k)}} = 1 + \frac{\partial F(\boldsymbol{x}^{(k)})}{\partial \boldsymbol{x}^{(k)}}.$$

Это значит, что даже насыщенный и полностью обученный слой F, производные которого близки к нулю, не помешает обучению.

Примеры таких «обходных путей», которые использовались в разных вариантах сети *ResNet* и других исследованиях этой группы авторов, показаны на рис. 5.8. В работе [242] проводится подробное сравнение нескольких вариантов остаточных блоков. Результаты оказались любопытными: остаточные блоки, которые теоретически должны быть более выразительными, на практике оказываются хуже, чем самые простые варианты.

Для примера мы изобразили на рис. 5.8, *в* очень простой вариант остаточного блока, в котором $\boldsymbol{y}^{(k)} = \boldsymbol{x}^{(k)} + F(\boldsymbol{x}^{(k)})$, а на рис. 5.8, *г* — вариант посложнее:

$$\boldsymbol{y}^{(k)} = \Big(1 - \sigma(f(\boldsymbol{x}^{(k)}))\Big)\, \boldsymbol{x}^{(k)} + \sigma(f(\boldsymbol{x}^{(k)}))F(\boldsymbol{x}^{(k)}),$$

где f — другая функция входа, реализованная через свертки 1×1. Это значит, что $F(\boldsymbol{x}^{(k)})$ и $\boldsymbol{x}^{(k)}$ суммируются не с равными весами, а с весами, управляемыми дополнительным «гейтом»[1]. Казалось бы, простой вариант является частным случаем сложного: достаточно просто обучить гейты так, чтобы веса были равными, то есть

[1] Особенно важны будут конструкции с такими управляющими гейтами для рекуррентных сетей — в разделе 6.3 гейты будут куда сложнее и интереснее.

чтобы всегда выполнялось $f(\boldsymbol{x}^{(k)}) = 0$. Однако эксперименты в [242] показали, что простота в данном случае важнее выразительности и важно обеспечить максимально свободное и беспрепятственное течение градиентов. На рис. 5.8, *д* показан вариант остаточного блока, который использовался в исходной статье [111], а на рис. 5.8, *е* — улучшенный вариант из [242]. Обратите внимание, что разница, по сути, только в том, что из «обходного пути» убрали ReLU-нелинейность, последнее «препятствие» на пути значений с предыдущего слоя.

Архитектурно все это приводит к тому, что становится возможным обучать очень, очень глубокие сети. Каймин Хе называет это «революцией глубины»: в VSS было 19 уровней, в GoogLeNet — 22 уровня, в первом варианте ResNet — сразу 152, а в последних версиях сетей с остаточными связями без проблем обучаются сети до тысячи уровней в глубину!

Это, безусловно, самые глубокие из реально используемых нейронных сетей. И они действительно работают: большинство лучших результатов в современных нейронных сетях используют в качестве распознавателя объектов разные варианты ResNet. Если вам нужно что-то распознавать на картинках, скорее всего, вы будете пользоваться одной из этих архитектур.

Правда, в некоторых приложениях от них отказываются ради скорости и экономии ресурсов: большая сверточная сеть с остаточными связями никак не поместится в смартфон. Если ресурсы важны, стоит посмотреть в сторону моделей, которые показывают немного более слабые результаты в собственно распознавании, но имеют при этом на порядок меньше весов; выделим, в частности, MobileNets [371] и SqueezeNet [504].

Впрочем, в заключение этого раздела отметим, что идея про гейт, управляющий остаточными блоками, как на рис. 5.8, *г*, все-таки оказалась довольно плодотворной. Именно она легла в основу так называемых *магистральных сетей (highway networks)*, предложенных группой Юргена Шмидхубера [506].

Идея магистральных сетей именно такая, как мы показывали выше: мы представляем $\boldsymbol{y}^{(k)}$, выход слоя k, как линейную комбинацию входа этого слоя $\boldsymbol{x}^{(k)}$ и результата $F(\boldsymbol{x}^{(k)})$, веса которой управляются другими преобразованиями:

$$\boldsymbol{y}^{(k)} = C(\boldsymbol{x}^{(k)})\boldsymbol{x}^{(k)} + T(\boldsymbol{x}^{(k)})F(\boldsymbol{x}^{(k)}),$$

где C — это гейт переноса (carry gate), а T — гейт преобразования (transform gate); обычно комбинацию делают выпуклой, $C = 1 - T$. Магистральные сети тоже позволили обучать очень глубокие сети с сотнями уровней [507], а затем эта конструкция была адаптирована для рекуррентных сетей [440].

Что здесь победит — простота или выразительность — вопрос пока открытый, но в любом случае варианты этих архитектур уже позволили сделать беспрецедентно глубокие сети, и останавливаться на достигнутом исследователи не собираются. А мы на этом завершаем краткий обзор современных сверточных архитектур. Пора двигаться дальше, к другой постановке задачи обучения — автокодировщикам.

5.5. Автокодировщики

> А спросите, для чего я так сам себя коверкал и мучил? Ответ: затем, что скучно уж очень было сложа руки сидеть; вот и пускался на выверты.
>
> *Ф. М. Достоевский. Записки из подполья*

В этом разделе мы познакомимся с одной из традиционных архитектур нейронных сетей, которая очень хорошо себя зарекомендовала в задачах *извлечения признаков*, когда нужно из сложных данных большой размерности выделить признаки, имеющие какой-то смысл с точки зрения того, что эти данные вообще собой представляют. Иными словами, мы будем решать задачу *обучения без учителя*: как извлечь из большого количества данных смысл, как найти закономерности, которые управляют этими данными, как понять, что в них общего, что они означают и как использовать их дальше?

Как и у многих других архитектур нейронных сетей, у автокодировщиков долгая и славная история. Впервые модель автокодировщика была представлена еще в 1986 году в классической работе Дэвида Румельхарта[1], Джеффри Хинтона и Рональда Уильямса, *Learning internal representations by error propagation* [459, 460]; с тех пор появилась масса различных вариантов автокодировщиков, но основная идея остается той же самой, и автокодировщики в целом только набирают популярность благодаря своей простоте и гибкости.

Итак, предположим, что у нас есть некий набор данных, который мы хотели бы описать с помощью новых «умных» признаков, которые были бы лучше и «интереснее» (о том, что это такое, мы поговорим ниже), чем исходное описание. Например, мы бы хотели описать рукописные цифры не пиксел за пикселом, а с помощью каких-то признаков, из которых было бы достаточно просто выяснить, какая цифра написана, или чьим почерком, или каким цветом... в общем, признаков, которые были бы более информативными. Это выглядит как задача обучения без учителя, которую трудно решать нейронными сетями: непонятно, как определить функцию ошибки для задачи «найти интересные признаки».

Основная идея автокодировщиков столь же проста, сколь и гениальна: давайте превратим задачу обучения без учителя в задачу обучения с учителем, сами себе придумаем тестовые примеры с известными правильными ответами. И сделаем мы это так: попросим модель обучиться выдавать на выходе ровно тот же пример, который подавали ей на вход! При этом она будет обучаться сначала создавать некое внутреннее представление, кодировать вход какими-то признаками, а потом декодировать их обратно, чтобы восстановить исходный вектор входов. Мы изобразили

---

[1] *Дэвид Румельхарт* (David E. Rumelhart, 1942–2011) — американский психолог и математик, один из основателей современной когнитивной науки. Он был одним из изобретателей алгоритма обратного распространения ошибки и создал целый ряд классических моделей процессов познания и обработки информации, впервые предложил научно проверяемые модели когнитивных процессов [410].

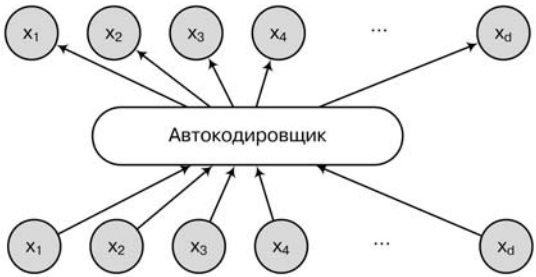


Рис. 5.9. Как работает автокодировщик

общую схему автокодировщика на рис. 5.9. Говоря чуть более формально, мы хотим обучить функцию $f(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{x}$, где $\boldsymbol{\theta}$ — это, как обычно, параметры нейронной сети.

Конечно, въедливый читатель спросит: в чем тут, собственно, проблема? Тоже мне бином Ньютона — скопировать вход в выход, для этого никакая оптимизация не нужна, вполне достаточно взять сеть с единичными матрицами весов, скопировать вход на скрытый слой, а потом скопировать скрытый слой на выход. И это действительно так: если не накладывать дополнительных ограничений на представление, которое должно получиться на скрытом слое, ничего умного не получится, и сеть будет обучаться просто копировать вход в выход. Искусство построения автокодировщиков, и в целом практически все содержание этой главы, состоит в том, чтобы придумать, какие ограничения и каким образом наложить на нейронную сеть, чтобы получающиеся на скрытых слоях признаки действительно оказались «интересными».

В классическом автокодировщике, который и предлагался в исходных работах [459, 460], дополнительное ограничение очень простое: чтобы на скрытом слое нельзя было просто скопировать вход, давайте уменьшим его размерность по сравнению с размерностью входа. Например, в разделе 3.6 мы кодировали рукописные цифры из датасета MNIST, которые представляют собой картинки размера 28×28 пикселов, в виде вектора длиной 784, по отдельной размерности для каждого пиксела. А теперь мы можем попробовать построить автокодировщик, у которого на скрытом слое будет, скажем, сто нейронов, попросив тем самым сеть представить каждую цифру из картинки 28×28 пикселов в виде вектора размерности 100.

Здесь, правда, возникает противоположный вопрос: почему это вообще может работать? Ведь отображение пространства большой размерности $\mathbb{R}^D$ в пространство меньшей размерности $\mathbb{R}^d$, $d < D$, не может быть взаимно однозначным, то есть мы всегда будем терять какую-то информацию; как же мы тогда будем восстанавливать исходные векторы $\boldsymbol{x} \in \mathbb{R}^D$?

И действительно, если бы нам нужно было научить модель отображать *все* возможные картинки размера 28×28 пикселов в пространство $\mathbb{R}^{100}$, это была бы невозможная задача: случайный вектор из пространства $\mathbb{R}^{784}$ содержит гораздо больше

информации, чем вектор из $\mathbb{R}^{100}$, и по сути мы не могли бы сделать ничего лучше, чем просто скопировать какие-нибудь сто координат исходного вектора, а остальные выбрать случайно.

К счастью, в реальных задачах с реальными данными кодировать случайные векторы не надо. Обычно оказывается, что хотя базовое пространство, в котором представлены данные, и имеет большую размерность, сами данные в нем лежат неподалеку от *многообразия* гораздо меньшей размерности. Нам вовсе не нужно кодировать любой белый шум из случайно зажженных 784 пикселов; нам нужно кодировать некие «осмысленные» изображения, которые обладают множеством очевидных для человека свойств: они выражают одну из десяти цифр, имеют определенную толщину линии, наклон, представляют собой обычно неразрывные, связные объекты или состоят из немногих штрихов. Все это очень сильно сужает множество возможных изображений цифр и позволяет надеяться на то, что существует представление рукописных цифр в пространстве меньшей размерности.

При таком подходе получается, что автокодировщик фактически решает задачу *понижения размерности* (dimensionality reduction): как отобразить большое пространство со сложными взаимосвязями в пространство более низкой размерности, где, будем надеяться, сложные взаимосвязи перейдут в зависимости попроще. Есть множество классических методов понижения размерности: анализ главных компонент[1] (principal component analysis, PCA) [1], сингулярное разложение матриц (singular value decomposition, SVD) [30], анализ независимых компонент (independent component analysis, ICA) [241] и многие другие.

Важное отличие того, что мы делаем сейчас, от классических методов понижения размерности в том, что нейронные сети могут выразить больше разных сложных многообразий, им не нужны настолько сильные предположения о структуре данных, как классическим моделям. Гибкость моделей, задаваемых нейронными сетями, часто позволяет выделить очень «интересные» признаки, и хотя модели становятся более сложными и хрупкими, очевидно, что именно это и требуется в реальной жизни: попробуйте-ка представить себе, как выглядит «многообразие рукописных цифр» в пространстве $\mathbb{R}^{784}$ или многообразие осмысленных текстов на русском языке в пространстве последовательностей букв...

На практике автокодировщики чаще используют для извлечения таких признаков из данных, которые в итоге позволяют уменьшить ошибку при последующем обучении с учителем. И оказывается, что в этом случае автокодировщики, в скрытом слое которых нейронов *больше*, чем во входном (их называют *overcomplete* autoencoders, а реально понижающие размерность — *undercomplete*), зачастую оказываются крайне полезными.

Кажется, что при такой архитектуре нейронной сети достаточно скопировать вход на произвольное подмножество нейронов скрытого слоя, а затем на выходные

[1] Кстати, об анализе главных компонент мы еще поговорим в разделе 10.3, а разделы 8.5 и 10.4 будут продолжением разговора об автокодировщиках.

нейроны, но нелинейная функция активации и регуляризация делают это практически невозможным. Например, редкий автокодировщик в наше время обходится без дропаута, но дропаут сильно мешает идее простого копирования; приходится все-таки «честно» выделять признаки.

Конечно, автокодировщики за двадцать лет своего существования получили сразу несколько серьезных «апгрейдов». «Обычные» автокодировщики, которые мы рассматривали выше, пытаются восстановить вход по нему же самому, то есть пытаются обучить тождественную функцию $f(\boldsymbol{x}) = \boldsymbol{x}$ либо с помощью средств, недостаточных, чтобы это выразить, либо с дополнительными ограничениями и целями, которые мешают просто взять и скопировать вход в выход. Тем не менее, по мере того как размерность скрытых слоев и выразительность модели будут расти — а мы ведь хотим, чтобы они росли, — мы будем все точнее восстанавливать вход по нему же самому, и справляться с потенциальным оверфиттингом будет все сложнее и сложнее.

Шумоподавляющий автокодировщик (denoising autoencoder) — это оригинальный и очень простой способ почти полностью избавиться от проблем оверфиттинга. Давайте будем восстанавливать не вход $\boldsymbol{x}$ по нему самому, а вход $\boldsymbol{x}$ по некоторому его *зашумленному* варианту $\tilde{\boldsymbol{x}}$. Например, давайте выберем 10 % пикселов изображения и заменим их нулями (черными пикселами) или случайными значениями интенсивностей, но восстановить при этом попросим не искаженный вариант картинки, а исходный, в котором все пикселы стоят на своих местах. Таким образом, автокодировщик должен будет не просто сжать полученный пример, но еще и частично восстановить утраченные в процессе зашумления данные, обучить не тождественную функцию $f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{x}$, как мы делали в этой главе раньше, а довольно сложную функцию $f(\tilde{\boldsymbol{x}}; \boldsymbol{\theta}) = \boldsymbol{x}$, которая уже неизбежно будет описывать многие интересные свойства поступающих на вход данных.

Кроме этого непосредственного преимущества появляются и другие. Во-первых, обратите внимание, что такой подход позволяет существенно увеличить объем обучающей выборки фактически бесплатно: мы ведь можем зашумлять один и тот же пример $\boldsymbol{x}$ по-разному, получая из него сразу много новых тренировочных примеров $\tilde{x}^1, \tilde{x}^2, \ldots, \tilde{x}^k$ с одним и тем же «правильным ответом» $\boldsymbol{x}$. Это, конечно, нельзя делать бесконечно: набор базовых тренировочных векторов все-таки не увеличивается, и если переборщить с генерацией их случайно зашумленных вариантов, получится тот же оверфиттинг; но в два-три раза датасет на практике обычно можно таким образом «увеличить». С другой стороны, зашумленность как регуляризатор заставляет автокодировщик пытаться выучивать независимые друг от друга признаки, ведь когда случайный шум ляжет по-другому, некоторых локальных признаков уже не будет, и придется по оставшимся восстанавливать утраченные.

Но пока это все были абстрактные рассуждения; как вносить случайный шум на практике? В реальных шупомодавляющих автокодировщиках почти всегда применяют один из двух способов зашумления входного сигнала:

- первый способ — это добавление ко входу случайного нормально распределенного шума с маленькой дисперсией, которая в данном случае и определяет уровень шума; такой подход хорошо работает для некоторых типов данных, но относительно редко применяется в обработке изображений, так что сейчас мы не будем подробно на нем останавливаться и оставим читателю простор для экспериментов;
- второй способ зашумления заключается в том, что часть входных нейронов попросту обнуляется; уровень шума здесь определяется тем, какая именно это часть.

Во втором способе, самом популярном в задачах обработки изображений, нейронная сеть, прогоняя очередной зашумленный пример через скрытый слой, по сути должна не просто «отфильтровать шум», а решить по ходу дела задачу восстановления утраченных данных. Благодаря этому фильтры, получающиеся в результате обучения шумоподавляющего автокодировщика, оказываются менее «шумными» и более локальными.

Другой важный вариант — *разреженные автокодировщики*. Начнем с небольшого лирического отступления. Одно из важных свойств настоящего человеческого мозга, которое наложило на нас очень серьезный отпечаток, — это огромное количество энергии, которое требуется мозгу. В человеческом теле мозг — главный ее потребитель, он тратит до 20–30 % энергии, которую мы получаем с пищей, при том, что по весу он составляет всего 2–3 % от общей массы тела. По правдоподобной версии, современный размер человеческого мозга (и не только физический размер, но и число нейронов и число уровней нейронов, которые во многом и определяют, насколько «умными» мы можем быть) эволюционно получился таким не только потому, что детей с такой большой головой было бы совсем трудно рожать, но и из энергетических соображений. Будь мозг еще больше, первым людям пришлось бы, при всем их глубоком уме, круглые сутки заниматься исключительно тем, чтобы непрерывно питаться, чтобы вообще хоть как-то поддерживать работу такого большого мозга.

Поэтому для человеческого мозга очень важно не просто обладать большими вычислительными ресурсами, но и задействовать их максимально эффективно. По меркам современных компьютеров мозг — чудо энергоэффективности: он использует всего лишь около 30 Вт мощности, чтобы управляться со всеми своими 10^{10} нейронами и 10^{14} синапсами. Одна из ключевых особенностей, позволяющих мозгу это делать, — это *разреженность* активации нейронов: в каждый момент времени в мозгу активна только очень небольшая часть нейронов. Если бы мозг представлял собой плотную модель и в каждый момент времени активировалась бы половина имеющихся нейронов, наши предки вряд ли долго протянули бы, ведь тогда не было ни чизбургеров, ни колы, ни даже обычного молочного шоколада.

Оказывается, что эта *разреженность* (sparsity) активаций — это желаемое свойство не только для мозга, для которого это было жизненно важно, но и для искусственных моделей. Иногда мы можем интерпретировать значение активации

нейрона именно как его активность, и тогда значения, близкие к единице, означают, что нейрон посылает импульсы, и наоборот, при близких к нулю значениях нейрон можно считать «погашенным», неактивным. В некоторых задачах мы хотели бы получить нейронную сеть, большинство нейронов которой неактивны в каждый момент времени. С одной стороны, это позволяет надеяться на то, что за различные «полезные» свойства, извлеченные из данных, отвечают отдельные, единичные нейроны; в отличие от ситуации, когда сразу большая группа нейронов отвечает за некоторое свойство, нам очень трудно интерпретировать их значения, и модель получается слишком хрупкой. Такой же эффект возникал в разделе 4.1 при дропауте, который можно считать своеобразным средством обеспечить разреженность.

Но как сделать нейронную сеть разреженной на практике? Давайте вернемся к вероятностной интерпретации активации нейронов. Для увеличения разреженности скрытого слоя нам бы хотелось, чтобы вероятность активации каждого отдельно взятого нейрона была низкой. Пусть нейрон представляет собой случайную величину с распределением Бернулли; проще говоря, нейрон — это обычная монетка, и среднее значение активации нейрона скрытого слоя соответствует вероятности получения единицы (например, решки) в испытании Бернулли. Пусть теперь желаемая вероятность активации нейрона равна ρ, а полученное из данных эмпирическое среднее равно $\hat{\rho}$. Расстояние Кульбака — Лейблера между модельным распределением и распределением данных равно

$$\mathrm{KL}\left(\rho \| \hat{\rho}\right) = \rho \log \frac{\rho}{\hat{\rho}}.$$

И теперь это расстояние, то есть меру непохожести между распределениями, можно просто добавить в целевую функцию как регуляризатор.

Итак, в этом разделе мы познакомились с основными идеями того, как создавать автокодировщики, важный метод извлечения признаков без учителя. Осталось только увидеть, как же все это работает на практике.

5.6. Пример: кодируем рукописные цифры

> В отчаянии я решил начать с чистого листа. Думать, как Алан Тьюринг. Обычно мы стараемся свести задачу к числам, а потом бросить на нее всю мощь математического анализа.
>
> *Н. Стивенсон. Криптономикон*

В этом разделе мы продолжаем серию примеров, связанных с распознаванием цифр на датасете MNIST; простите, что столь однообразны наши примеры, но на самом деле полезно посмотреть, как работают самые разные модели на одном и том же наборе данных, а MNIST — это классический датасет, на котором все эти модели работают достаточно быстро и хорошо. Мы начнем с того, что рассмотрим

обычный избыточный автокодировщик, а потом будем постепенно модифицировать полученный код и получать другие варианты автокодировщиков.

Сначала импортируем все, что нам понадобится:

```
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

И загрузим набор данных MNIST:

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Как мы уже обсуждали, датасет состоит из 70 000 размеченных черно-белых изображений, которые разделены между тренировочной, тестовой и валидационной выборками по 55 000, 10 000 и 5000 примеров соответственно.

Зададим гиперпараметры сети:

```
batch_size = 64
latent_space = 128
learning_rate = 0.1
```

Давайте для простоты эксперимента опишем однослойный автокодировщик. Для этого объявим его веса

```
ae_weights = {"encoder_w": tf.Variable(
                  tf.truncated_normal([784, latent_space], stddev=0.1)),
              "encoder_b": tf.Variable(
                  tf.truncated_normal([latent_space], stddev=0.1)),
              "decoder_w": tf.Variable(
                  tf.truncated_normal([latent_space, 784], stddev=0.1)),
              "decoder_b": tf.Variable(
                  tf.truncated_normal([784], stddev=0.1))}
```

и зададим тензоры:

```
ae_input = tf.placeholder(tf.float32, [batch_size, 784])
hidden = tf.nn.sigmoid(
    tf.matmul(ae_input, ae_weights["encoder_w"]) + ae_weights["encoder_b"])
visible_logits = tf.matmul(
    hidden, ae_weights["decoder_w"]) + ae_weights["decoder_b"]
visible = tf.nn.sigmoid(visible_logits)
```

На самом деле тензор `visible` для обучения не нужен, так как мы будем использовать стандартную функцию ошибки из библиотеки TensorFlow, `sigmoid_cross_entropy_with_logits`. Однако позже он пригодится нам для того, чтобы визуализировать восстановленные изображения. А сама функция ошибки для автокодировщика в виде перекрестной энтропии запишется так:

```
ae_cost = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(fc_v_logits, ae_input))
```

Выберем алгоритм оптимизации из TensorFlow, в данном случае AdaGrad:

```
optimizer = tf.train.AdagradOptimizer(learning_rate)
ae_op = optimizer.minimize(ae_cost)
```

Как и раньше, нам остается только инициализировать переменные, создать сессию и запустить процесс обучения:

```
sess = tf.Session()
sess.run(tf.initialize_global_variables())

for i in xrange(10000):
    x_batch, _ = mnist.train.next_batch(batch_size)
    sess.run(ae_op, feed_dict={ae_input: x_batch})
```

После обучения на 10 000 мини-батчей ошибка восстановления на тестовых данных составляет примерно 0,22, а после 100 000 падает до 0,12 и при дальнейшем обучении может быть улучшена до значений, меньших чем 0,1. Изображения, которые получаются после восстановления, все еще выглядят как цифры, но очень сильно «размыты».

Чтобы сделать разреженный автокодировщик, нужно, как мы обсуждали выше, добавить регуляризатор. Зададим в качестве дополнительных параметров ρ и β вес регуляризации в функции стоимости:

```
rho = 0.05
beta = 1.0
```

Определим теперь тензор для регуляризационного слагаемого:

```
data_rho = tf.reduce_mean(hidden, 0)
reg_cost = - tf.reduce_mean(tf.log(data_rho/rho) * rho +
    tf.log((1-data_rho)/(1-rho)) * (1-rho))
```

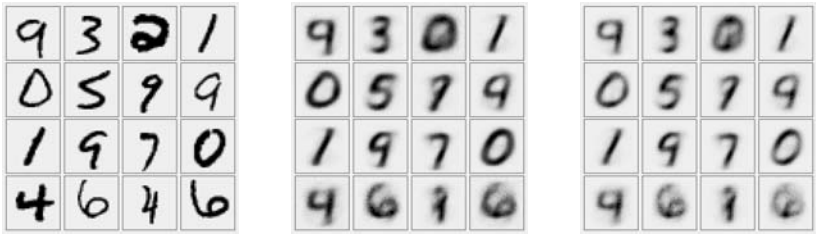
Здесь для оценки $\hat{\rho}$ мы используем среднее значение активации скрытого слоя по мини-батчу для каждого нейрона, а для упрощения вычислений переворачиваем дроби внутри логарифмов. Общая функция стоимости теперь выглядит так:

```
total_cost = ae_cost + beta * reg_cost
```

И именно она передается оптимизатору вместо `ae_cost`.

Посмотрите внимательно, что здесь произошло: мы задали дополнительное слагаемое для функции ошибки; оно тем меньше, чем ближе распределение активации нейронов к придуманному нами распределению, в котором монетка выпадает орлом (то есть нейрон активируется) с вероятностью $\rho = 0{,}05$. Иными словами, мы просто попросили модель обучиться так, чтобы на каждом отдельном входе на скрытом слое активировалось примерно 5 % нейронов; а параметр β отвечает за то, насколько убедительно мы ее об этом попросили.

После нескольких миллионов мини-батчей и нескольких эпох обучения модели средние значения нейронов скрытого слоя уменьшатся и станут близки к ρ.



а *б* *в*

Рис. 5.10. Пример работы разреженного автокодировщика: *а* — исходные изображения; *б* — реконструированные; *в* — восстановленные из «обрезанного» скрытого слоя

Но где же обещанная разреженность? Ведь хоть значения активаций и уменьшились, они все так же остаются ненулевыми, а нам для улучшения эффективности модели неинтересно иметь околонулевые веса, нам нужны строгие, «жесткие» нули. Давайте проведем небольшой эксперимент: возьмем случайный мини-батч, спроецируем его в вектор активаций скрытого слоя и обнулим все элементы, меньшие 0,1. Поскольку в наших экспериментах $\rho = 0{,}05$, это весьма значительная часть элементов скрытого слоя. Это само по себе вполне ожидаемо, но самое интересное здесь то, что оставшихся элементов будет вполне достаточно для восстановления исходных изображений! Разреженный автокодировщик не обманул: мы действительно обучили скрытый слой так, что теперь можно обнулить все маленькие веса и все еще получить хорошее приближение к данным.

Чтобы это нарисовать, введем дополнительный «зашумленный скрытый слой» и его декодировщик:

```
noised_hidden = tf.nn.relu(hidden - 0.1) + 0.1
noised_visible = tf.nn.sigmoid(
    tf.matmul(noised_hidden, ae_weights["decoder_w"]) + ae_weights["decoder_b"])
```

Мы не будем использовать его в функции ошибки, а только для отрисовки картинок; полученные рукописные цифры изображены на рис. 5.10, *в*, и в сравнении с «честно» реконструированными цифрами на рис. 5.10, *б* видно, что разница не так уж и велика.

Кстати, такой трюк с обнулением весов можно использовать не только в специально для этого созданных разреженных автокодировщиках. Во многих современных глубоких моделях регуляризация устроена так, что, обнуляя редко или слабо используемые веса, можно значительно сократить скрытый слой и упростить применение модели (но не ее обучение). В частности, эта идея используется для того, чтобы уместить обученные глубокие модели на мобильных устройствах.

Еще одно замечательное свойство разреженного автокодировщика заключается в том, что «фильтры», получаемые в результате обучения, часто имеют явную и понятную семантику. В нашем случае оказывается, что большинство фильтров

учится распознавать различные очертания цифр. Но как мы смогли это заметить? Как понять не то, какие нейроны соответствуют данной картинке (для этого нужно просто запустить нейронную сеть), а наоборот: какие картинки соответствуют данному нейрону?

Чтобы это проиллюстрировать, давайте возьмем отдельный нейрон внутреннего слоя и попробуем подобрать такие входы, которые максимизировали бы его активацию. Так как функция активации скрытого слоя $\sigma(a)$ монотонна от своего аргумента a, ее максимальное значение будет достигаться там же, где максимального значения достигает ее параметр. Значение i-го нейрона скрытого слоя до применения нелинейности вычисляется как $\boldsymbol{x}^\top W_{*i} + b_i$, где параметр b_i, хоть и влияет на само значение максимума, не меняет точку, в которой он достигается. В таком случае задача сводится к тому, чтобы максимизировать скалярное произведение

$$\boldsymbol{x}^\top W_{*i} = \sum_j x_j W_{ij}.$$

Заметим, что, поскольку мы имеем дело с изображениями, значения пикселов которых принадлежат отрезку $[0, 1]$, очевидно, что максимума эта сумма достигает в случае, когда $x_j = 1$, если $W_{ij} > 0$, и $x_j = 0$ в противном случае.

Однако на такой «бинарной» картинке точки, имеющие очень маленький, но положительный вес, будут прорисованы так же сильно, как и самые важные, и мы вряд ли сможем хорошо их отличить друг от друга. Поэтому давайте наложим дополнительное ограничение на значения входных нейронов, нормализуем их: $\sum_i x_i^2 = 1$. Тогда максимальная активация на выходе i-го нейрона будет достигаться в следующей точке[1]:

$$x_i = \frac{W_{ij}}{\text{norm}(W_{*j})}.$$

Мы изобразили некоторые из 128 фильтров разреженного автокодировщика на рис. 5.11. Хотя выглядят все они довольно зашумленно (все-таки это самая простая из возможных архитектур), видно, что многие из них содержат явные очертания цифр.

Для того чтобы реализовать шумоподавляющий автокодировщик и протестировать его на MNIST, достаточно внести минимальные изменения в уже написанный код. Начнем с того, что добавим еще одну заглушку для «испорченных» данных:

```
noisy_input = tf.placeholder(tf.float32, [batch_size, 784])
```

Конечно, эта заглушка должна иметь такой же размер, как и `ae_input`, так как функция потерь после сжатия и восстановления примеров будет подсчитываться

[1] Хотя эту задачу можно очень просто решить с помощью метода λ-множителей Лагранжа, объяснение этого метода выходит за рамки данной книги, и мы предоставляем знакомому с базовым курсом математического анализа читателю возможность самому получить ответ.

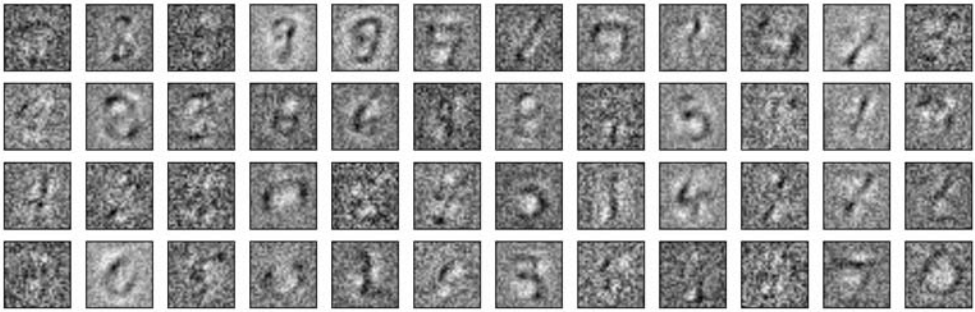


Рис. 5.11. Некоторые фильтры разреженного автокодировщика

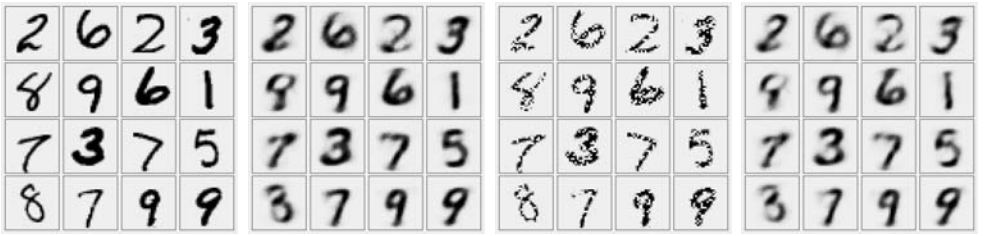
на основе исходных данных, а не зашумленных. Скрытый слой теперь тоже формируется с использованием новой заглушки:

```
hidden = tf.nn.sigmoid(tf.matmul(noisy_input, ae_weights["encoder_w"])
                       + ae_weights["encoder_b"])
```

А когда нам нужно будет для тестирования полученного автокодировщика прогнать через скрытый слой «чистые» данные, без шума, мы просто передадим исходный батч в `noisy_input`. Основной цикл обучения меняется незначительно:

```
for i in xrange(updates):
    x_batch, _ = mnist.train.next_batch(batch_size)
    noise_mask = np.random.uniform(0., 1., [batch_size, 784]) < noise_prob
    noisy_batch = x_batch.copy()
    noisy_batch[noise_mask] = 0.0
    sess.run(ae_op, feed_dict={ae_input: x_batch, noisy_input: noisy_batch})
```

Функция `np.random.uniform` возвращает массив со значениями, выбранными (сэмплированными) из случайной величины, равномерно распределенной на отрезке, который задается первыми двумя параметрами; в нашем случае это отрезок [0,1]. Третий параметр задает размер массива. Обратите внимание на то, что перед обнулением элементов входных данных мы копируем батч целиком, вызывая `x_batch.copy()`; если бы мы этого не сделали, а использовали обычное присваивание, то в результате обнулились бы значения и в `x_batch`. Для эксперимента мы выбрали вероятность «выкидывания» пиксела `noise_prob = 0.3`. Эта вероятность на первый взгляд кажется очень большой: как так, мы выкидываем 30 % картинки? да что там вообще останется? Оказывается, что на практике наилучшие результаты получаются как раз в случаях, когда уровень шума очень, на первый взгляд чрезмерно, большой: просить модель угадывать «закрытую» треть, а то и половину картинки оказывается правильной стратегией. На рис. 5.12 изображены примеры результатов работы шумоподавляющего автокодировщика: видно, что реконструкция уже вполне адекватная, да и с зашумленными входами он справляется хорошо.



а *б* *в* *г*

Рис. 5.12. Пример работы шумоподавляющего автокодировщика: *а* — исходные изображения; *б* — реконструированные; *в* — зашумленные; *г* — восстановленные из зашумленных

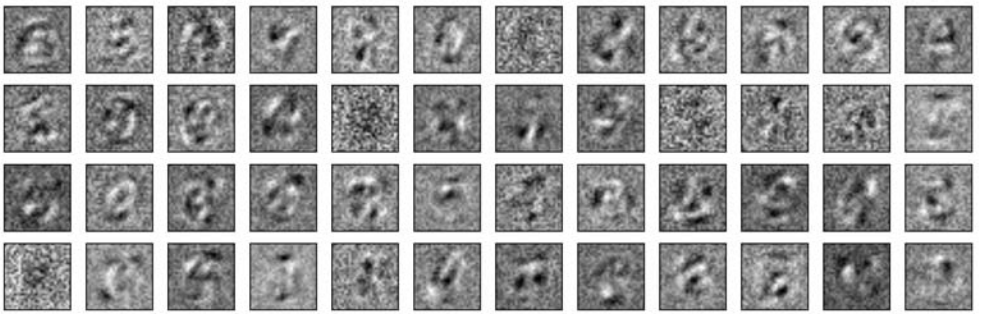


Рис. 5.13. Некоторые фильтры шумоподавляющего автокодировщика

А на рис. 5.13 мы изобразили активации фильтров шумоподавляющего автокодировщика, сделанные по тому же принципу, что на рис. 5.11. Видно, что хотя качество реконструкции у шумоподавляющего автокодировщика не уступает разреженному, активации фильтров устроены по-другому: среди них, как и следовало ожидать, меньше «готовых» цифр и больше отдельных частей, штрихов, из которых можно затем собрать то или иное изображение.

Итак, мы увидели в действии обычные, разреженные и шумоподавляющие автокодировщики. А теперь, чтобы диалектически завершить эту главу синтезом двух ее основных тем, давайте вернемся к сверточным сетям и попробуем реализовать *сверточный автокодировщик*.

Как он будет работать? Об автокодировщиках мы уже многое знаем: суть их в том, чтобы сначала построить некое внутреннее представление на внутреннем слое нейронов, а потом «развернуть» его обратно, реконструировать вход на выходе. Мы до сих пор все время рассматривали сверточные слои, которые берут на вход некую «картинку» и применяют к ней сверточные фильтры, получая представление в виде сначала совсем локальных, а потом все более и более глобальных

признаков. Это отлично подходит для построения первой половины автокодировщика, от входа до внутреннего представления.

Но как же потом разворачивать? В случае полносвязной сети мы просто строили такую же, симметричную архитектуру для декодирования; нам было не так уж важно, какие будут размеры у матрицы весов: 784×128 или наоборот. Но со свертками не все так очевидно.

Чтобы ответить на этот вопрос, давайте введем операцию *деконволюции*, или транспонированной свертки. Каждый фильтр в сверточном слое можно представить как операцию, сжимающую область $k \times k$ в одно число; с другой стороны, мы можем определить и обратную операцию — развернуть одно число в матрицу $k \times k$. Нужно только аккуратно учесть шаг свертки и дополнение нулями, но по сути мы можем просто транспонировать сверточный тензор и получить деконволюцию.

Разумеется, в TensorFlow такая операция уже реализована. Как всегда, начинаем с импорта библиотек и загрузки набора данных:

```
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
batch_size, learning_rate = 64, 0.01
```

Веса автокодировщика задаются как для обычной сверточной сети:

```
ae_weights = {
    "conv": tf.Variable(tf.truncated_normal([5, 5, 1, 4], stddev=0.1)),
    "b_hidden": tf.Variable(tf.truncated_normal([4], stddev=0.1)),
    "deconv": tf.Variable(tf.truncated_normal([5, 5, 1, 4], stddev=0.1)),
    "b_visible": tf.Variable(tf.truncated_normal([1], stddev=0.1))
}
```

Напомним, что первые две размерности тензора задают размер фильтра, а третья и четвертая, соответственно, число фильтров в текущем слое и в следующем. На первый взгляд, удивительно: почему декодирующая часть имеет те же размерности, что и кодирующая? Разве не нужно транспонировать матрицы? Причина здесь чисто техническая: TensorFlow сама транспонирует веса и применит их так, как надо; единственное, что нам для этого будет нужно, — это дополнительно передать в TensorFlow тензор с размерами результирующего слоя.

```
input_shape = tf.pack([batch_size, 28, 28, 1])
```

Поскольку мы реализуем автокодировщик, то выходной слой должен иметь такой же размер, как и входной. Теперь мы готовы определить все необходимые для обучения тензоры:

```
ae_input = tf.placeholder(tf.float32, [batch_size, 784])
images = tf.reshape(ae_input, [-1, 28, 28, 1])
hidden_logits = tf.nn.conv2d(ae_input, ae_weights["conv"],
                  strides=[1, 2, 2, 1], padding="SAME") + ae_weights["b_hidden"]
```

```
hidden = tf.nn.sigmoid(conv_h_logits)

visible_logits = tf.nn.conv2d_transpose(hidden,
              ae_weights["deconv"],
              input_shape, strides=[1, 2, 2, 1],
              padding="SAME") + ae_weights["b_visible"]
visible = tf.nn.sigmoid(visible_logits)
```

Самый простой способ правильно сделать деконволюцию — передать те же параметры, что и при свертке. При этом параметры `strides` и `padding` обозначают то же самое, что в свертке, с той лишь разницей, что характеризуют теперь выходной слой, а не входной.

В качестве целевой функции мы снова используем перекрестную энтропию:

```
optimizer = tf.train.AdagradOptimizer(learning_rate)
conv_cost = tf.reduce_mean(
       tf.nn.sigmoid_cross_entropy_with_logits(conv_v_logits, images))
conv_op = optimizer.minimize(conv_cost)
```

Осталось только создать сессию и инициализировать переменные:

```
init = tf.initialize_global_variables()
sess = tf.Session()
sess.run(init)
```

И можно запускать обучение!

```
for i in xrange(100000):
    x_batch, _ = mnist.train.next_batch(batch_size)
    sess.run(conv_op, feed_dict={ae_input: x_batch})
```

Через некоторое время ошибка на тестовой выборке падает примерно до значения 0,09, что меньше, чем при полносвязном автокодировщике. Казалось бы, что тут удивительного: мы применили более сложную архитектуру, специально придуманную для обработки таких данных, как картинки, и получили результат лучше, чем в полносвязном автокодировщике. Но если присмотреться внимательнее, результат сверточных сетей покажется куда более удивительным.

Ведь что мы имеем на самом деле в сверточной сети в этом примере: четыре сверточных фильтра размером 5×5 каждый, столько же весов для декодирующей части, плюс еще по одному весу для свободного члена скрытого и видимого слоев. Итого 205 весов. А полносвязный автокодировщик, с которым мы сравниваем результаты, состоял из 784×196 весов кодировщика и стольких же для декодировщика, плюс веса свободных членов; если все сложить, получится аж 308 308 весов! Кажется логичным, что нейронная сеть большего размера должна обучиться лучше, но мы видим, что, хотя большая сеть действует вполне адекватно, не расходится и действительно обучается делать то, что надо, маленькая и юркая сверточная сеть ее все равно побеждает. В чем же тут дело?

Теоретически говоря, действительно, полносвязная нейронная сеть способна как минимум повторить результат сверточной сети: ей ничто не мешает имитировать веса, полностью аналогичные сверткам, у нее есть вполне достаточно степеней свободы и для этого, и для еще более сложных конструкций. Однако проблема полносвязной сети в том, что когда параметров обучения становится на три порядка больше, то и времени для обучения требуется намного больше, а главное, с любой, даже самой сильной регуляризацией, необходимо намного больше данных. С учетом того, что примеров в обучающей выборке ограниченное число, каждый из них приходится использовать для обучения многократно, что может привести и в конце концов приводит к переобучению.

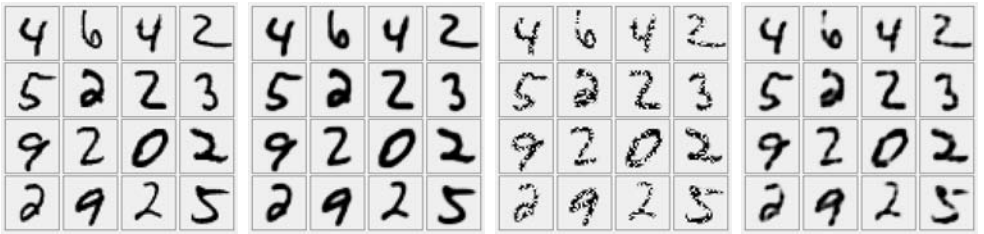
Собственно, саму архитектуру сверточных слоев можно считать формой регуляризации, которая особенно удачно подходит для обработки данных, обладающих некоей пространственной структурой: одномерные свертки хорошо подходят для обработки звука или других временных рядов, двумерные — изображений, трехмерные или комбинации двумерных и одномерных — для видео и т. д. Это и понятно: основная идея сверточных сетей, которую мы уже обсуждали, состоит в том, чтобы выделять одни и те же признаки по всей «длине» или «площади» данных. Для изображений здесь важно и то, что пикселы изображения сильно связаны друг с другом локально, а активация фильтров на таких особенностях изображения, как границы объектов или определенные формы, действительно несет в себе большое количество информации, и то, что собственно конкретное место изображения, в котором находится выделенный признак, обычно не столь важно: сдвиг на пару пикселов точно не должен приводить к плохим результатам, а сверточные сети как раз выделяют одни и те же признаки во всех окнах сразу.

Все это приводит к тому, что при решении практических задач, связанных с обработкой изображений и компьютерным зрением, сверточные сети представляют собой намного более эффективную альтернативу полносвязным.

Раз уж сверточная сеть требует такого маленького числа весов, давайте сделаем ее глубже и добавим еще один сверточный слой. Для этого нужно задать веса для нового сверточного слоя:

```
ae_weights = {
    "conv1": tf.Variable(tf.truncated_normal([5, 5, 1, 4], stddev=0.1)),
    "b_conv1": tf.Variable(tf.truncated_normal([4], stddev=0.1)),
    "conv2": tf.Variable(tf.truncated_normal([5, 5, 4, 16], stddev=0.1)),
    "b_hidden": tf.Variable(tf.truncated_normal([16], stddev=0.1)),
    "deconv1": tf.Variable(tf.truncated_normal([5, 5, 4, 16], stddev=0.1)),
    "b_deconv": tf.Variable(tf.truncated_normal([4], stddev=0.1)),
    "deconv2": tf.Variable(tf.truncated_normal([5, 5, 1, 4], stddev=0.1)),
    "b_visible": tf.Variable(tf.truncated_normal([1], stddev=0.1)),
}
```

А затем завести тензор для размера промежуточного слоя, который будет использоваться при деконволюции:



а *б* *в* *г*

Рис. 5.14. Пример работы сверточного автокодировщика: *а* — исходные изображения; *б* — реконструированные; *в* — зашумленные; *г* — восстановленные из зашумленных

```
h1_shape = tf.pack([batch_size, 14, 14, 4])
```

Заново описываем граф сети:

```
images = tf.reshape(ae_input, [-1, 28, 28, 1])

conv_h1_logits = tf.nn.conv2d(images, ae_weights["conv1"],
                    strides=[1, 2, 2, 1], padding="SAME") + ae_weights["b_conv1"]
conv_h1 = tf.nn.relu(conv_h1_logits)

hidden_logits = tf.nn.conv2d(conv_h1, ae_weights["conv2"],
                    strides=[1, 2, 2, 1], padding="SAME") + ae_weights["b_hidden"]
hidden = tf.nn.relu(hidden_logits)

deconv_h1_logits = tf.nn.conv2d_transpose(hidden,
                    ae_weights["deconv1"],
                    h1_shape, strides=[1, 2, 2, 1],
                    padding="SAME") + ae_weights["b_deconv"]
deconv_h1 = tf.nn.relu(deconv_h1_logits)

visible_logits = tf.nn.conv2d_transpose(deconv_h1,
                    ae_weights["deconv2"],
                    input_shape, strides=[1, 2, 2, 1],
                    padding="SAME") + ae_weights["b_visible"]
visible = tf.nn.sigmoid(visible_logits)
```

Все остальное можно оставить без изменений. В этот раз мы использовали 1025 весов, и уже примерно после 10 000 итераций обучения ошибка на тестовых данных падает до 0,08, а в результате продолжительного обучения останавливается на уровне около 0,07. Качество восстановленных изображений тоже заметно улучшилось — их можно увидеть на рис. 5.14. Теперь размытости практически не осталось, и можно сказать, что MNIST мы таким сверточным автокодировщиком окончательно «победили», примерно так же, как раньше в этой главе достигли сверточной сетью очень высокой точности классификации.

Итак, в этой очень важной главе мы разобрались и с основными концепциями сверточных сетей, и с автокодировщиками. Сверточные сети — это одна из важнейших архитектур нейронных сетей, которая использует сверточные фильтры с общими весами, применяя тем самым одни и те же преобразования к разным частям входа. Такая экстремальная регуляризация позволяет сверточным сетям обучаться очень эффективно и быть весьма глубокими даже «из коробки», а такие более современные идеи, как остаточные связи, позволяют обучать нейронные сети практически неограниченной глубины. Сейчас мы уже на полпути к тому, чтобы закончить рассмотрение основных базовых архитектур и перейти к применениям, улучшениям и прочим новомодным идеям. Однако сначала нам нужно поговорить о другом столпе современных нейронных сетей — сетях *рекуррентных*.

Глава 6
Рекуррентные нейронные сети,

или Как правильно кусать себя за хвост

TL;DR

В этой главе мы познакомимся с рекуррентными нейронными сетями, которые могут сохранять скрытое состояние от одного входа к следующему. В частности, мы:

- замотивируем рекуррентные сети разнообразными задачами обработки последовательностей;
- познакомимся с основными архитектурами рекуррентных нейронных сетей;
- узнаем о проблеме затухающих градиентов...
- ...и тут же решим ее с помощью карусели константной ошибки в LSTM и GRU;
- рассмотрим несколько возможностей добиться того же в «обычных» рекуррентных сетях;
- разберем большой пример посимвольного порождения текстов.

6.1. Мотивация: обработка последовательностей

> Сделайте же наконец то, к чему вас так тянет, будьте последовательны. А там увидите.
>
> *А. Камю*

> Не стремиться к мировому господству — на это у меня не хватило духу. Меня избивали.
>
> *Б. Брехт. Разговоры беженцев*

В предыдущих разделах мы говорили о нейронных сетях, которые получают на вход некоторый вектор данных и пытаются по нему предсказать тот или иной результат. Вектор этот должен для данной архитектуры сети иметь одну и ту же размерность. В результате получается, что сеть может постоянно делать одну и ту же операцию: предсказывать выход по входам, причем рассматривать каждый следующий вход совершенно независимо от предыдущего.

Однако жизнь, конечно же, устроена сложнее. Часто исходными данными для решения нашей задачи являются *последовательности*. Это могут быть временные ряды (изменения цен акций, показания датчиков), естественно возникающие последовательности с зависимыми элементами (предложения естественного языка, человеческая речь при распознавании) — словом, любые данные, где соседние точки зависят друг от друга, и эту зависимость нельзя игнорировать.

Конечно, можно пытаться моделировать последовательности с внутренними зависимостями и обычными нейронными сетями. Например, одна из первых приходящих в голову идей — зафиксировать некоторую длину истории l и подавать на вход нейронной сети предыдущие l значений ряда: будем пытаться предсказать значение x_n из ряда $x_1,x_2,\ldots,x_{n-2},x_{n-1}$ как функцию $x_n = f(x_{n-1},\ldots,x_{n-l})$.

Пример такой архитектуры показан на рис. 6.1. Здесь выходы получаются из трех предыдущих входов: $y_4 = f(x_1,x_2,x_3)$, $y_5 = f(x_2,x_3,x_4)$ и $y_6 = f(x_3,x_4,x_5)$. Если задачей модели является предсказание следующего элемента (такая задача стоит, например, при порождении текстов или предсказании временных рядов), в качестве функции ошибки можно взять разницу между y_j и соответствующим x_j (например, процент угаданных слов). А если задача состоит в том, чтобы предсказать одну последовательность по другой (как, например, в распознавании речи: получив на вход последовательность звуков, выдать последовательность слов), нам потребуются правильные ответы, последовательность будет выглядеть как $\{x_i, t_i\}_{i=1}^n$, а функция ошибки будет оценивать, насколько точно наш результат y_j предсказывает правильный ответ t_i из тренировочной выборки.

Заметим, что веса у всех изображенных сетей общие, то есть такая сеть будет рассматривать последовательность из тренировочных данных как много независимых тестовых примеров. Фактически это одномерные свертки.

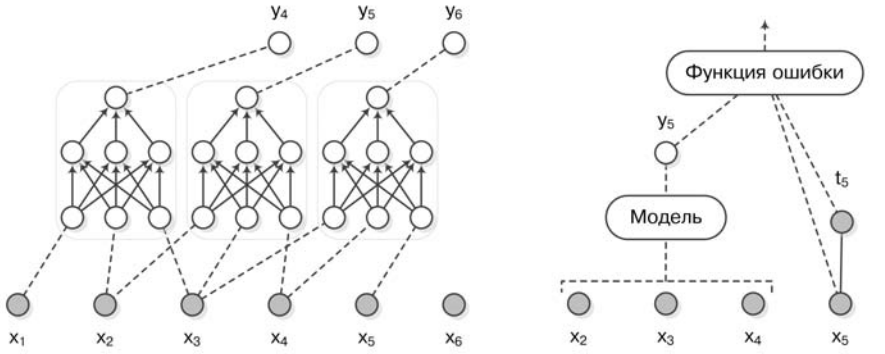


Рис. 6.1. Архитектура обычной нейронной сети с фиксированным размером истории. *Слева*: одна и та же нейронная сеть применяется к последовательным окнам входа. *Справа*: результат нейронной сети сравнивается с очередным элементом последовательности

Для некоторых задач такая идея может сработать, но часто длина зависимости тоже не известна заранее, а предсказание получить, тем не менее, нужно в любой момент времени. Попробуйте ответить: сколько предыдущих слов нужно запомнить, чтобы уверенно предсказать следующее слово в тексте? Может быть, контекст полностью описан тремя предыдущими словами, а может быть, тремя тысячами слов; это никак нельзя определить раз и навсегда для любого текста. В такой ситуации было бы хорошо, если бы нейронная сеть могла *запоминать* что-то из истории приходящих на вход данных, сохранять некое внутреннее состояние, которое можно было бы потом использовать для предсказания будущих элементов последовательности.

Для того чтобы отразить такую временную зависимость в данных, часто используются так называемые *рекуррентные нейронные сети*. «Обычные» нейронные сети — описанные ранее многослойные перцептроны — имеют фиксированное число входов и воспринимают каждый из них как независимый. В рекуррентных же сетях связи между нейронами могут идти не только от нижнего слоя к верхнему, но и от нейрона к «самому себе», точнее, к предыдущему значению самого этого нейрона или других нейронов того же слоя. Именно это позволяет отразить зависимость переменной от своих значений в разные моменты времени: нейрон обучается использовать не только текущий вход и то, что с ним сделали нейроны предыдущих уровней, но и то, что происходило с ним самим и, возможно, другими нейронами на предыдущих входах. Мы проиллюстрировали эту идею на рис. 6.2. Он отличается от рис. 6.1 только наличием связей между последовательными элементами; эти связи отражают присутствие некоторого *скрытого состояния* s_t, которое во время t формально зависит от всего, что раньше происходило во входной последовательности.

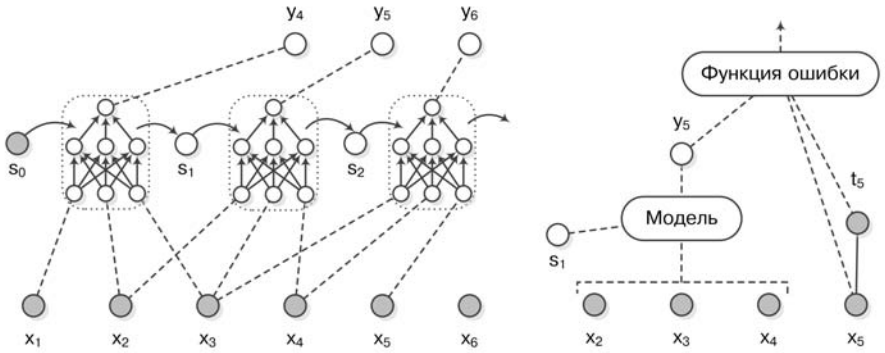


Рис. 6.2. Архитектура рекуррентной нейронной сети в тех же обозначениях, что на рис. 6.1. *Слева*: рекуррентная нейронная сеть получает на вход свое предыдущее состояние и последовательные окна входа. *Справа*: результат рекуррентной нейронной сети точно так же сравнивается с очередным элементом последовательности

Как достаточно широко употреблявшуюся историческую альтернативу такому подходу стоит упомянуть *нейронные сети с временной задержкой* (time-delay neural networks, TDNN), которые известны еще с 80-х годов XX века [293, 420] (любопытно, что и в этих статьях без Джеффри Хинтона не обошлось). Они тоже предназначены для обработки последовательностей, но при этом сети с временной задержкой — не рекуррентные сети, а обычные, очень похожие на сверточные, только свертка здесь происходит по времени, а не по размерности входа.

Суть в том, что входы сети, поступающие в виде последовательности, подаются на вход сети не по одному, а последовательно с задержками; такую картину мы уже видели на рис. 6.1, разница в случае TDNN только в том, что задержки могут быть разными, а в некоторых вариантах даже автоматически настраиваться [572]. Однако это все же не «настоящая» рекуррентная архитектура, и дальше мы ее рассматривать не станем.

В этой главе мы будем подробно говорить о рекуррентных нейронных сетях, но сначала давайте посмотрим, какими, собственно, бывают разные задачи, связанные с обработкой последовательностей; в качестве дополнительного источника мы здесь всецело рекомендуем широко известный пост Андрея Карпатого [273]. Посмотрите на рис. 6.3: на нем схематично показаны основные типы задач машинного обучения, связанных с последовательностями. По характеру входов и выходов задачи можно выделить такие пять вариантов:

- один вход, один выход (one-to-one, рис. 6.3, *а*); здесь последовательности как бы и нет, мы просто должны независимо обработать каждый элемент входов и получить соответствующий выход, никакие скрытые состояния никуда не передаются; однако заметим, что работа с последовательностью как на

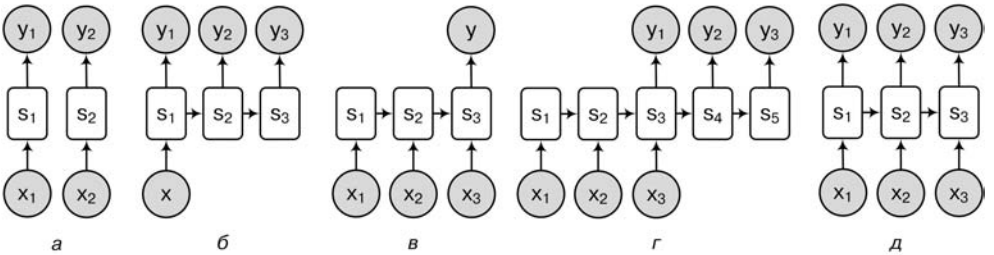


Рис. 6.3. Задачи с последовательностями: *а* — один вход, один выход; *б* — один вход, последовательность выходов; *в* — последовательность входов, один выход; *г* — последовательность входов, затем последовательность выходов; *д* — синхронизированные последовательности входов и выходов

рис. 6.1, когда окна обрабатываются независимо, вполне укладывается в эту схему, просто входы получатся достаточно сильно зависимыми;

- один вход, последовательность выходов (one-to-many, рис. 6.3, *б*); здесь мы должны «развернуть» вход, который сам по себе не имеет структуры последовательности, в последовательность выходов; например, аннотирование картинок представляет собой такую задачу: на входе картинка, на выходе текст (последовательность);
- последовательность входов, один выход (many-to-one, рис. 6.3, *в*); в этот тип задач укладываются любые задачи классификации последовательностей; например, анализ тональности (sentiment analysis): по данному тексту (то есть последовательности) выдать, положительно или отрицательно он окрашен;
- последовательность входов, затем последовательность выходов (many-to-many, рис. 6.3, *г*); здесь речь идет о том, чтобы «свернуть» входную последовательность, закодировать ее неким скрытым состоянием, а потом «развернуть» это скрытое состояние обратно в уже совершенно другую последовательность; например, по этой общей схеме работают системы машинного перевода (вход — предложение на одном языке, выход — на другом) и диалоговые системы (вход — реплика собеседника, выход — своя собственная реплика);
- синхронизированные последовательности входов и выходов (synchronized many-to-many, рис. 6.3, *д*); а здесь нужно снабдить своей меткой каждый элемент последовательности, но в отличие от рис. 6.3, *а* есть смысл также переносить на следующий временной шаг некое скрытое состояние; например, представьте, что нам нужно разметить видеопоток, в котором каждый последующий кадр, конечно, представляет собой самостоятельную картинку, но она обычно очень похожа на предыдущую и следующую.

6.2. Распространение ошибки и архитектуры RNN

> Естественная история в XVII–XVIII вв. была... совокупностью правил, позволяющей группировать высказывания в ряды и цепочки, в совокупности неустранимых схем зависимости, порядка и последовательности, где перераспределялись и получали концептуальную значимость различные рекуррентные элементы.
>
> *М. Фуко. Археология знания*

Хотя в принципе рекуррентные сети продолжают классические идеи достаточно прямолинейно, нам теперь становится сложно просто взять и применить алгоритм обратного распространения ошибки. И действительно, рекуррентная архитектура представляется очень удачной для обработки последовательностей, а последовательности — это и тексты, и видео, и музыка... но как же теперь считать градиенты?

В прямой нейронной сети ошибка на конкретном нейроне вычисляется как функция от ошибок нейронов, которые используют его выходное значение. Но что делать в случае, когда нейрон принимает в качестве входа результат вычисления в нем самом? Да и как, собственно, провести само вычисление, если в нем участвует его результат? Иначе говоря, до сих пор у нас все графы вычислений были *ациклическими* (без циклов, то есть без «замкнутых кругов»), а теперь получается, что в графе сети одни сплошные циклы? Как можно вычислить функцию, которая принимает сама себя на вход?!

На самом деле, конечно, эта петля в графе вычислений нам только кажется. Мы действительно используем результаты вычисления той же функции, но это результаты вычисления функции на *предыдущих* шагах. В результате вычисление, которое делает рекуррентная сеть, можно развернуть обратно до начала последовательности. Например, обозначая в примере на рис. 6.2 состояние сети после i-го тестового примера через $s_i = h(x_i, x_{i+1}, x_{i+2}, s_{i-1})$, получим, что y_6 на самом деле вычисляется так:

$$y_6 = f(x_3, x_4, x_5, s_2) = f(x_3, x_4, x_5, h(x_2, x_3, x_4, s_1)) = \\ = f(x_3, x_4, x_5, h(x_2, x_3, x_4, h(x_1, x_2, x_3, s_0))).$$

Здесь уже s_0 взять неоткуда, это будет начальное состояние сети. А параметры сети здесь спрятаны внутри функций f и h, то есть в этой формуле одни и те же веса встречаются много раз.

Можно сказать, что на каждом шаге сеть создает несколько копий самой себя. Каждая из этих копий принимает на вход текущее окно в определенный момент времени (определенную часть последовательности) и значение, полученное из предыдущей копии, затем каким-то образом их комбинирует и передает получившийся результат в следующий элемент. Таким образом, на каждом шаге мы фактически обучаем глубокую нейронную сеть, в которой столько слоев, сколько

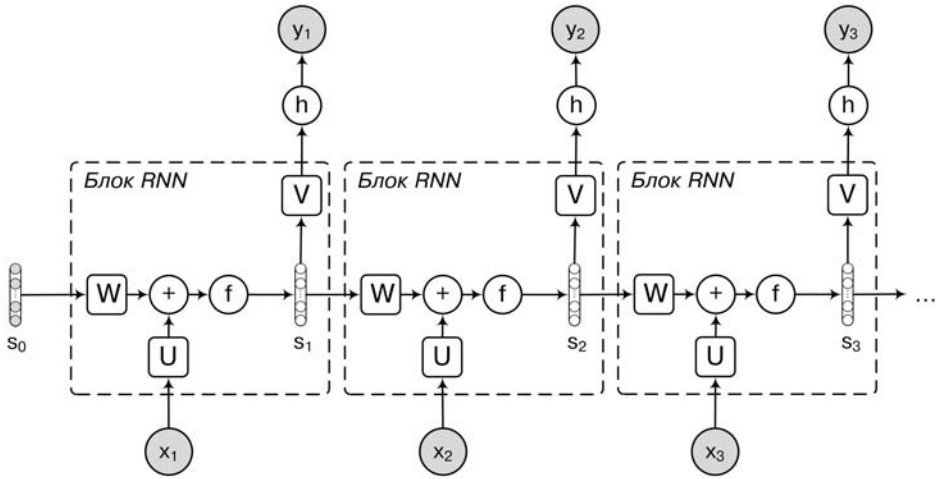


Рис. 6.4. Архитектура «простой» рекуррентной нейронной сети

элементов в последовательности мы уже видели. Основное ее отличие от обычной нейронной сети, которые мы рассматривали до сих пор, состоит в том, что веса на каждом слое одинаковые, все слои делят одни и те же переменные между собой (shared weights).

Теперь можно и подсчитать градиент в развернутом виде; мы это сделаем на конкретном примере, но сначала все-таки придется все строго формально доопределить. Для этого примера мы возьмем одну из самых простых архитектур рекуррентной сети: будем использовать нейроны с сигмоидом в качестве функции активации и считать, что все переходы, кроме этого сигмоида, линейные, а на выходе мы решаем задачу классификации с помощью softmax-функции от выхода сети или просто применяем ту или иную функцию активации h.

Обозначим через W матрицу весов для перехода между скрытыми состояниями, через U матрицу весов для входов, а через V — для выходов. Архитектура простой рекуррентной нейронной сети (да, иногда именно так и пишут, Simple RNN) показана на рис. 6.4. Входы и скрытые состояния будем считать векторами. Кроме того, чтобы формулы были не слишком громоздкими, будем считать, что на вход всегда подается один вектор $\boldsymbol{x}$; это не приводит ни к какой потере общности: например, в формуле выше можно было бы просто переобозначить $\boldsymbol{x}_i = (x_1,x_2,x_3)$. В итоге мы получаем такое формальное задание сети — в момент времени t:

$$\begin{aligned} \boldsymbol{a}_t &= \boldsymbol{b} + W\boldsymbol{s}_{t-1} + U\boldsymbol{x}_t, \quad & \boldsymbol{s}_t &= f(\boldsymbol{a}_t), \\ \boldsymbol{o}_t &= \boldsymbol{c} + V\boldsymbol{s}_t, & \boldsymbol{y}_t &= h(\boldsymbol{o}_t), \end{aligned}$$

где f — это нелинейность собственно рекуррентной сети (обычно σ, tanh или ReLU), а h — функция, с помощью которой получается ответ (например, softmax).

На рисунке мы, как обычно, не стали изображать свободные члены $\boldsymbol{b}$ и $\boldsymbol{c}$, чтобы не загромождать картинку, они подразумеваются в стрелочках. Собственно блоком RNN называется часть от входов до вычисления $\boldsymbol{o}_t$; а слой RNN — это блок, развернутый во времени на входную последовательность (или несколько блоков, которые делят друг с другом веса, тут уж как посмотреть).

Будем считать, что прошло T шагов, $t = 1..T$, и у нас задана какая-то функция ошибки $L(\boldsymbol{o}_1, \ldots, \boldsymbol{o}_T)$. Теперь можно напрямую попытаться подсчитать частные производные, постепенно «разворачивая» внутренние состояния $\boldsymbol{s}_t$ от $t = T$ обратно к $t = 1$. Получится, что один и тот же вес из матрицы W участвует в этом процессе $T - 1$ раз, по числу переходов, но полностью развернутый граф все равно будет, конечно, ациклическим. Получается, что рекуррентная сеть — это как будто очень-очень многоуровневая обычная сеть, в которой одни и те же веса переиспользуются на каждом уровне. Такие общие веса имеют ряд приятных свойств. Во-первых, для их хранения достаточно одной матрицы, все T «слоев» в данном случае сугубо виртуальные, и веса нужно хранить только один раз. Во-вторых, общие веса позволяют избегать некоторых проблем глубоких сетей: градиенты по весам не затухают до нуля сразу же.

Однако у рекуррентных сетей есть и свои проблемы. Во-первых, градиенты, конечно, сами по себе не затухают, но зато они могут *взорваться* (exploding gradients): если матрица весов такова, что заметно увеличивает норму вектора градиента при проходе через один «виртуальный слой» обратного распространения, получится, что при проходе через T слоев эта норма возрастет экспоненциально от T, веса ведь одни и те же.

Во-вторых, хотя градиенты не затухают совсем, влияние текущего входа или текущего состояния сети, тем не менее, обычно не может распространяться слишком далеко. Влияние текущего входа затухает экспоненциально по мере удаления. Это серьезная проблема, которая не позволяет «обычным» рекуррентным сетям обучаться распознавать далекие зависимости в данных.

Например, можно попытаться обучить рекуррентную сеть читать тексты (это мы будем делать в этой книге и на практике — в разделе 6.6 мы построим довольно простую, но вполне настоящую языковую модель), но сеть, скорее всего, сможет обучиться только тому, как слова или буквы влияют на своих близких соседей, а о том, чтобы понять, как слово повлияет на происходящее через 20–30 слов, в следующих предложениях или даже следующих абзацах, и мечтать не приходится. Чтобы обучать более долгосрочные зависимости, нам потребуется составлять рекуррентные сети не из «обычных» нейронов, а из существенно более сложных «кирпичиков»; это мы начнем делать с раздела 6.3, но сначала обсудим еще несколько общих архитектур, в которые эти «кирпичики» можно будет подставить.

Итак, теперь мы знаем, как проводить градиентный спуск в рекуррентных нейронных сетях. Но и это еще не все. Мы пока что обсудили разные способы того, как скрытые состояния, входы и выходы рекуррентной сети могут связываться друг

с другом и зависеть друг от друга. Но, как дотошные читатели уже наверняка хотят нам напомнить, это книга о *глубоком* обучении — а глубоких рекуррентных сетей мы пока не рассматривали вовсе! Все эти конструкции пока что использовали только один уровень нейронов, просто делали это несколько раз в разных контекстах, для моделирования разных функций.

К счастью, получить из подобных конструкций глубокие сети совсем несложно; для этого нужно просто заменить какие-нибудь из частей рекуррентной архитектуры многослойными конструкциями. Формально это просто значит, что мы по-другому, более сложным образом представляем некоторые из функций, составляющие рекуррентную сеть. Однако, как вы уже догадались, есть много разных способов это сделать, и они могут пригодиться в разных ситуациях. Не будем вдаваться в подробности, но кратко перечислим возможные подходы на основе самой простой архитектуры, изображенной на рис. 6.4.

1. Можно представить глубокой сетью функцию от входа к скрытому слою; интуиция здесь в том, что глубокие представления могут помочь рекуррентной сети понять временную структуру между последовательными временными шагами, а затем уже можно будет представить полученные соотношения между выделенными абстрактными признаками в виде более простых функций, приближая их более простыми нейронными сетями.

Такой подход к базовым рекуррентным сетям использовался, например, в распознавании речи [75]; а блестящую иллюстрацию сути и интуиции этого подхода мы увидим в главе 7, где будем обучать распределенные представления слов в естественном языке. Мы увидим, как, если правильно организовать обучение абстрактных признаков, можно получить разумные соотношения даже между такими сложными объектами, как слова естественного языка, на уровне простых линейных соотношений [137].

2. Второй вариант — смоделировать глубокой сетью функцию от скрытого слоя на выходы. Здесь интуиция в том, что на скрытом слое сеть может начать обучать и распознавать очень сложные и хитрым образом запутанные между собой факторы, которые, возможно, не так уж прямо относятся к тем переменным, которые мы пытаемся предсказывать, и их нужно «распутать» обратно, прежде чем делать предсказания.

Один из возможных вариантов такого подхода может состоять в том, чтобы выходной слой рекуррентной сети заменить не глубокой сетью, а какой-нибудь другой моделью, тоже более сложной и выразительной, чем одноуровневая сеть. Например, ограниченные машины Больцмана именно так, как часть рекуррентной сети, применялись для порождения полифонической музыки в работе [56].

3. Третий логичный вариант — моделировать глубокой сетью функцию перехода между скрытыми слоями. В [425] этот подход применяли для разметки изображений: там переходы рекуррентной сети моделировались сверточной сетью, а затем эта идея была расширена на гауссовские процессы [283].

В этом подходе, правда, возникает больше проблем, чем в других: как мы уже обсуждали, обратное распространение во времени — это вычислительно достаточно сложный процесс, который и с одноуровневыми-то связями требует некоторой изобретательности, а если моделировать такой переход глубокой сетью, вычислительные проблемы становятся совсем сложными; однако оказывается, что их можно до некоторой степени преодолеть за счет добавления связей «напрямую» между несколькими шагами (shortcut connections) [433].

4. Заключительный, четвертый вариант наконец-то предлагает что-то новенькое: вместо того чтобы моделировать отдельные компоненты рекуррентной сети глубокими сетями, можно просто рассмотреть всю рекуррентную сеть целиком как слой и использовать ее выходы как входы для следующего слоя. Это очень мощная идея, и мы будем ею постоянно пользоваться; эффект здесь состоит в том, что в результате каждый слой достаточно естественным образом действует в своем собственном «масштабе времени», примерно как каждый слой сверточной сети действует в своем масштабе, на свой размер окна входов. Данная идея используется очень часто, она появилась еще в начале 1990-х, до современной революции глубокого обучения [138, 192, 258, 472].

Еще один важный вариант RNN (в любом их изводе) — это *двунаправленные* рекуррентные сети. Дело в том, что часто бывает так, что RNN к концу последовательности уже забывают, с чего там начиналось; и вообще последние элементы последовательности, даже если мы не забудем начало, всегда будут гораздо важнее первых и в обычной RNN, и в сети из LSTM или GRU-ячеек (к которым мы перейдем в разделе 6.3).

Поэтому часто рассматривают так называемые *двунаправленные* рекуррентные сети (bidirectional RNN). Давайте для входной последовательности запустим RNN (обычно с разными весами) два раза: один слой будет читать последовательность слева направо, а другой — справа налево. Матрицы весов абсолютно независимы, между ними нет взаимодействия, просто для каждого элемента последовательности получатся два состояния: слева направо и справа налево. Разумеется, это работает только для последовательностей, которые даны нам сразу целиком (как предложения естественного языка).

Все это проиллюстрировано на рис. 6.5, где слева изображена структура обычной рекуррентной нейронной сети, а справа, на рис. 6.5, *б*, — двунаправленной. На этой схеме мы «спрятали» матрицы W и U в один блок и сконцентрировались на том, чтобы детально показать, что происходит с выходами; связи, относящиеся ко идущей справа налево рекуррентной сети, на рис. 6.5, *б* показаны пунктиром. Формально говоря, в двунаправленной сети мы вычисляем состояния s_t слева направо и состояния s'_t справа налево, а затем сливаем их в один результат уже на уровне выхода; это значит, что выход вычисляется как

$$\begin{aligned} \boldsymbol{s}_t &= \sigma\left(\boldsymbol{b} + W\boldsymbol{s}_{t-1} + U\boldsymbol{x}_t\right), & \boldsymbol{s}'_t &= \sigma\left(\boldsymbol{b}' + W'\boldsymbol{s}'_{t+1} + U'\boldsymbol{x}_t\right), \\ \boldsymbol{o}_t &= \boldsymbol{c} + V\boldsymbol{s}_t + V'\boldsymbol{s}'_t, & \boldsymbol{y}_t &= h\left(\boldsymbol{o}_t\right). \end{aligned}$$

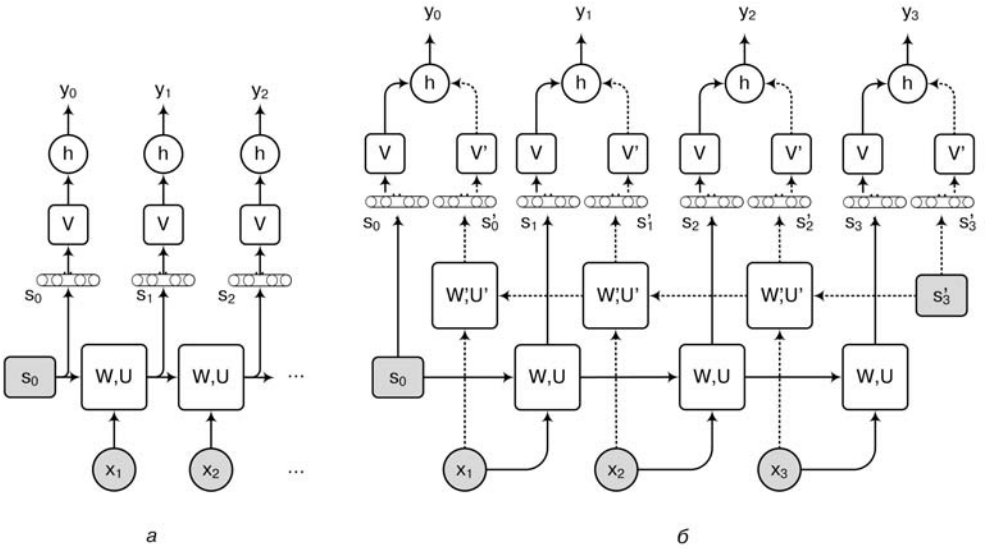


Рис. 6.5. Рекуррентные нейронные сети: *а* — обычная рекуррентная сеть; *б* — двунаправленная рекуррентная сеть; пунктиром изображены связи, относящиеся к сети, обрабатывающей входную последовательность справа налево

Вместо классической рекуррентной сети из трех матриц, конечно, может стоять любая другая конструкция; на практике обычно используют двунаправленные LSTM или GRU, с которыми мы познакомимся в следующих параграфах.

Важно понимать, что мотивация двунаправленных сетей состоит не в том, чтобы просто получить два разных скрытых состояния на всю последовательность. Если бы это было все, чего мы хотели, то было бы непонятно, почему мы запускаем сеть именно слева направо и справа налево, ведь можно было бы, например, сойтись в середине — чем хуже? Мотивация в том, чтобы получить состояние, отражающее контекст и слева, и справа *для каждого элемента последовательности*. Поэтому основные примеры задач, где двунаправленные сети лучше обычных, не в том, чтобы генерировать текст (когда нужно предсказать следующий элемент по предыдущим), а в том, чтобы, скажем, размечать слова по частям речи (ведь там важно посмотреть на все предложение целиком, и слева, и справа от слова). Например, в машинном переводе двунаправленная рекуррентная сеть будет работать вместе с механизмом внимания, чтобы получать хорошие векторы состояний для каждого слова, а не для всего предложения в целом.

Итак, мы познакомились с основными архитектурами рекуррентных сетей. Сейчас RNN (в основном основанные на LSTM и GRU ячейках, о которых речь пойдет ниже) используются очень широко, и в литературе можно выделить два противоположных тренда. С одной стороны, рекуррентные сети иногда возникают

даже там, где это не обязательно: для многих задач RNN слишком мощны, и модели становятся слишком сложными без нужды. Часто RNN применяют просто потому, что это hot new thing, и даже при в общем-то не сильно улучшающемся результате модель, основанная на рекуррентных сетях, скорее всего привлечет больше внимания (грубо говоря, легче будет опубликовать статью). С другой стороны, есть некоторая тенденция к замещению рекуррентных сетей сверточными: часто оказывается, что сверточные сети могут сделать то же самое, а обучать их при этом гораздо проще; мы увидим несколько таких примеров, когда будем говорить об обработке естественного языка. Но все равно именно рекуррентные сети остаются основной архитектурой для обработки последовательностей. О том, что им позволило добиться таких успехов и чем современные рекуррентные сети отличаются от классических, известных с конца 1980-х годов, мы и поговорим дальше.

6.3. LSTM

> Мой интерес к Великому Беспорядку (хаосу), конечно же, вызван неожиданным приходом старости, о которой я узнал по трем признакам: потере кратковременной памяти, приобретению долговременной памяти и желанию написать книгу.
>
> *Т. Лири. Семь языков Бога*

Как мы уже говорили в разделе 6.2, обычные рекуррентные сети очень плохо справляются с ситуациями, когда нужно что-то «запомнить» надолго: влияние скрытого состояния или входа с шага t на последующие состояния рекуррентной сети экспоненциально затухает. Что же делать? Решения, которые на данный момент предлагаются в глубоком обучении, состоят главным образом в том, чтобы изменить, усложнить архитектуру одного «кирпичика» рекуррентной сети. Оказывается, что вместо одного-единственного числа, на которое влияют все последующие состояния, можно сконструировать специального вида ячейку, в которой мы сможем явным образом смоделировать в том или ином виде «долгую память», процессы записи и чтения из этой «ячейки памяти» и так далее. Конечно, у такой ячейки будет не один набор весов, как у обычного нейрона, а сразу несколько, и обучение станет сложнее, но на практике часто оказывается, что оно того стоит.

Одна из самых широко известных и часто применяющихся конструкций таких ячеек — это LSTM (от слов *Long Short-Term Memory*; на русский язык это можно перевести как «долгая краткосрочная память», но в живом употреблении, конечно, русского перевода аббревиатуры LSTM мы ни разу не слышали) [336]. В обучении глубоких сетей LSTM используется постоянно, и мы еще много раз встретим в этой книге сети из LSTM-ячеек.

Стандартная архитектура LSTM-ячейки показана на рис. 6.6. В LSTM есть три основных вида узлов, которые называются *гейтами*[1]: *входной* (input gate), *забывающий* (forget gate) и *выходной* (output gate), а также собственно *рекуррентная ячейка* со скрытым состоянием. Кроме того, в LSTM часто добавляют еще так называемые *замочные скважины* (peepholes) — дополнительные соединения, которые увеличивают связность модели (о них мы поговорим ниже).

Формально говоря, если обозначить через $\boldsymbol{x}_t$ входной вектор во время t, через $\boldsymbol{h}_t$ — вектор скрытого состояния во время t, через W_i (с разными вторыми индексами) — матрицы весов, применяющиеся ко входу, через W_h — матрицы весов в рекуррентных соединениях, а через $\boldsymbol{b}$ — векторы свободных членов, мы получим следующее формальное определение того, как работает LSTM: на очередном входе $\boldsymbol{x}_t$, имея скрытое состояние из предыдущего шага $\boldsymbol{h}_{t-1}$ и собственно состояние ячейки $\boldsymbol{c}_{t-1}$, мы последовательно вычисляем

$$
\begin{aligned}
\boldsymbol{c}'_t &= \tanh\left(W_{xc}\boldsymbol{x}_t + W_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{c'}\right) && \textit{candidate cell state}\\
\boldsymbol{i}_t &= \sigma\left(W_{xi}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right) && \textit{input gate}\\
\boldsymbol{f}_t &= \sigma\left(W_{xf}\boldsymbol{x}_t + W_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right) && \textit{forget gate}\\
\boldsymbol{o}_t &= \sigma\left(W_{xo}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right) && \textit{output gate}\\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{c}'_t, && \textit{cell state}\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t) && \textit{block output}
\end{aligned}
$$

Выглядит весьма запутанно для одного-единственного звена сети. Мы проиллюстрировали LSTM-ячейку на рис. 6.6, где на рис. 6.6, *а* вводится обозначение гейта, а на рис. 6.6, *б* показана собственно структура LSTM; пунктирные линии введены исключительно для удобства и показывают связи, относящиеся к скрытому состоянию $\boldsymbol{h}_t$ (а не $\boldsymbol{c}_t$). Но даже картинка выглядит довольно сложно. Давайте пошагово разберемся, что значат эти формулы.

На вход LSTM, как и в «обычной» RNN, подаются два вектора: новый вектор из входных данных $\boldsymbol{x}_t$ и вектор скрытого состояния $\boldsymbol{h}_{t-1}$, который получен из скрытого состояния этой ячейки на предыдущем шаге. Кроме того, внутри у каждого LSTM-блока есть «ячейка памяти» (cell) — вектор, который выполняет функцию памяти. Вектор ячейки на шаге t мы обозначили выше через $\boldsymbol{c}_t$, а $\boldsymbol{c}'_t$, который получается в первом же уравнении, — это вектор, полученный из входа и предыдущего скрытого состояния, который становится кандидатом на новое значение памяти. Получается он из $\boldsymbol{x}_t$ и $\boldsymbol{h}_{t-1}$ весьма обычным для нейронных сетей преобразованием: сначала линейная функция, потом гиперболический тангенс, все как в обычных нейронных сетях.

---

[1] И снова мы калькируем терминологию. Здесь у нас больше оправданий, чем обычно, — термин «гейт» уже устоялся в схемной сложности, когда речь идет о схемах, вычисляющих ту или иную функцию. Впрочем, в советской литературе 1960–1970-х годов (а многие классические результаты схемной сложности были получены именно в СССР) употреблялся термин «вентиль», но мы, пожалуй, не рискнем его использовать.

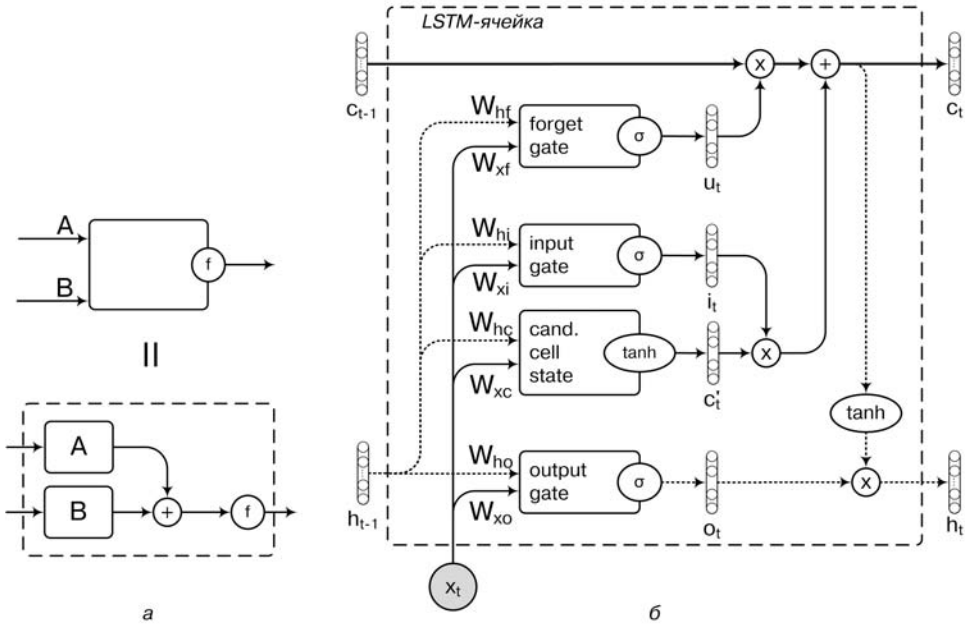


Рис. 6.6. LSTM: *а* — обозначение гейта с двумя входами; *б* — структура LSTM-ячейки

Но $\boldsymbol{c}'_t$ — это всего лишь кандидат в новое значение памяти. Прежде чем его запишут на место $\boldsymbol{c}_{t-1}$, значение-кандидат и старое значение проходят через еще два гейта: входной гейт $\boldsymbol{i}_t$ и забывающий гейт $\boldsymbol{f}_t$. Посмотрите на формулу

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{c}'_t.$$

Здесь новое значение получается как линейная комбинация из старого с коэффициентами из забывающего гейта $\boldsymbol{f}_t$ и нового кандидата $\boldsymbol{c}'_t$ с коэффициентами из входного гейта $\boldsymbol{i}_t$. Там, где значения вектора забывающего гейта $\boldsymbol{f}_t$ будут близки к нулю, старое значение $\boldsymbol{c}_{t-1}$ «забудется», а там, где значения $\boldsymbol{i}_t$ будут велики, новый входной вектор прибавится к тому, что было в памяти.

Обратите внимание: покомпонентное умножение приводит к тому, что на очередном шаге может быть перезаписана только часть «памяти» LSTM-ячейки; и какая это будет часть, тоже определяет сама ячейка в зависимости от того, что получается на выходах забывающего гейта $\boldsymbol{f}_t$ и входного гейта $\boldsymbol{i}_t$. И еще более того, поскольку эта линейная комбинация «мягкая» и по дороге все пропускается через сигмоиды σ, LSTM-ячейка может не просто выбрать, записать новое значение или выкинуть его, а еще и сохранить любую линейную комбинацию старого и нового значения, причем коэффициенты могут быть разными в разных компонентах вектора; и эти решения ячейка принимает в зависимости от конкретного входа. Все

это делает LSTM-ячейки весьма гибкими; а если теперь вспомнить, что таких ячеек у нас много, из них состоит рекуррентная сеть, то становится понятно, почему LSTM-ячейки оказались настолько успешной модификацией.

Но это все пока были абстрактные рассуждения: мы бы хотели, чтобы гибкость архитектуры LSTM позволяла обучать долгосрочные зависимости, и теоретически она, конечно, может это сделать. Однако архитектура обычных рекуррентных сетей тоже теоретически позволяет обучить все что угодно, но на практике так не получается; чем же LSTM принципиально отличается от обычных RNN?

Дело в том, что LSTM благодаря своей архитектуре решают проблему исчезающих градиентов, которая мешала рекуррентным сетям обучать долгосрочные зависимости. Чтобы лучше увидеть разницу, давайте на время представим себе LSTM без забывающего гейта (собственно, именно в таком виде LSTM и появился изначально, в самых ранних работах Шмидхубера); в наших обозначениях будем считать, что $\boldsymbol{f}_t = 1$ во всех компонентах и для всех t. Тогда вектор «памяти» ячейки будет вычисляться как

$$\boldsymbol{c}_t = \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{c}'_t.$$

А это значит, что

$$\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = 1.$$

Этот эффект, при котором в рекурсивном вычислении состояния ячейки нет никакой нелинейности, в литературе называется «каруселью константной ошибки» (constant error carousel): ошибки в сети из LSTM пропагируются без изменений, и скрытые состояния LSTM могут, если сама ячейка не решит их перезаписать, сохранять свои значения неограниченно долго. Это решает проблему «исчезающих градиентов»: независимо от матрицы рекуррентных весов ошибка сама собой затухать не будет. Отметим, впрочем, что с другой стороны LSTM никак не защищает от «взрывающихся градиентов»: если градиент начнет неограниченно расти, линейная зависимость от $\boldsymbol{c}_{t-1}$ никак этому не помешает. Поэтому в жизни реализации LSTM, как и RNN, обычно используют отсечение градиентов (gradient clipping), искусственно запрещая им расти далее определенных значений.

Важный практический момент: хотя обычно веса нейронной сети инициализируются маленькими случайными числами и это прекрасно работает для почти всех весов LSTM-ячеек, особым случаем является свободный член забывающего гейта $\boldsymbol{b}_f$. Дело в том, что если этот свободный член инициализировать около нуля, это фактически будет значить, что все LSTM-ячейки изначально будут иметь значение $\boldsymbol{f}_t$ около $1/2$. А это значит, что та самая карусель константной ошибки перестает работать: мы начинаем с того, что фактически вводим во все ячейки фактор «забывания» в $1/2$, и в результате ошибки и память будут затухать экспоненциально. Поэтому свободный член $\boldsymbol{b}_f$ нужно инициализировать большими значениями, около 1 или даже 2: тогда значения забывающих гейтов $\boldsymbol{f}_t$ в начале обучения будут близки к нулю и градиенты будут вольно литься по просторам нашей рекуррентной архитектуры.

И еще одно небольшое техническое замечание: может показаться, что для вычисления, например, $\boldsymbol{c}'_t$ нужно сначала умножить матрицу W_{xc} на вектор $\boldsymbol{x}_t$, потом умножить матрицу W_{hc} на вектор $\boldsymbol{h}_{t-1}$, а потом сложить результаты. Это несколько противоречило бы идеям векторизации и параллелизации всего на свете, но, к счастью, эту операцию тоже можно представить в виде умножения матриц:

$$W_{xc}\boldsymbol{x}_t + W_{hc}\boldsymbol{h}_{t-1} = \begin{pmatrix} W_{xc} \\ W_{hc} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1}. \end{pmatrix}$$

Более того, все, что происходит в первых четырех формулах, можно представить в виде умножения одной большой матрицы на один вектор, в котором $\boldsymbol{x}_t$ и $\boldsymbol{h}_{t-1}$ просто будут размножены по четыре раза (мы также предполагаем, что уже «поглотили» векторы свободных членов $\boldsymbol{b}$ в один из векторов $\boldsymbol{x}$ или $\boldsymbol{h}$):

$$\begin{pmatrix} W_{xc} & & & & W_{hc} & & & \\ & W_{xi} & & & & W_{hi} & & \\ & & W_{xf} & & & & W_{hf} & \\ & & & W_{xo} & & & & W_{ho} \end{pmatrix} \begin{pmatrix} \boldsymbol{x}_t \\ \boldsymbol{x}_t \\ \boldsymbol{x}_t \\ \boldsymbol{x}_t \\ \boldsymbol{h}_{t-1} \\ \boldsymbol{h}_{t-1} \\ \boldsymbol{h}_{t-1} \\ \boldsymbol{h}_{t-1} \end{pmatrix} = \begin{pmatrix} W_{xc}\boldsymbol{x}_t + W_{hc}\boldsymbol{h}_{t-1} \\ W_{xi}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1} \\ W_{xf}\boldsymbol{x}_t + W_{hf}\boldsymbol{h}_{t-1} \\ W_{xo}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1} \end{pmatrix}.$$

Таким образом, вычисление всех четырех векторов $\boldsymbol{c}'_t$, $\boldsymbol{i}_t$, $\boldsymbol{f}_t$, $\boldsymbol{o}_t$ можно параллелизовать и записать в векторном виде; это удобно и эффективно.

Заметим, что обычную рекуррентную сеть можно рассматривать как частный случай LSTM с гейтами, где некоторые значения зафиксированы в виде констант. Если мы установим забывающий гейт $\boldsymbol{f}_t$ всегда равным нулю во всех компонентах, а во входном гейте $\boldsymbol{i}_t$ и выходном гейте $\boldsymbol{o}_t$ установим все компоненты в единицы, получится, что мы всегда забываем предыдущее значение ячейки $\boldsymbol{c}_{t-1}$, запоминаем значение-кандидат $\boldsymbol{c}'_t$ в чистом виде, и на выход после этого оно переходит целиком. Единственным отличием от стандартных рекуррентных сетей будет в таком случае «лишнее» применение tanh.

Итак, мы подробно рассмотрели базовую, стандартную конфигурацию LSTM-ячейки. Сама архитектура LSTM имеет, по меркам нейросетевой науки, очень долгую историю: ее впервые разработали Зепп Хохрайтер и Юрген Шмидхубер еще в середине 1990-х годов [227, 228]. В тех работах уже были сети из ячеек, содержащих входные и выходные гейты, но не было забывающих гейтов; они добавились в самом конце 1990-х [178], сначала для задач, в которых нужно обнулять свое скрытое состояние (например, обучение грамматик), а потом уже и в целом для задач, где нужна более долгая и лучше управляемая память. Кроме того, в ранних работах об LSTM полное обратное распространение ошибки во времени делалось только для состояний ячеек $\boldsymbol{c}_t$, а градиенты по другим рекуррентным весам обрезались после первого же шага; только в 2005 году в работе [193] обратное распространение во времени было применено ко всем весам, что позволило получить хорошие результаты для распознавания фонем.

Есть еще одна очень часто встречающаяся модификация LSTM; она появилась в работе Герса и Шмидхубера 2000 года [177] и с тех пор стала частью большинства стандартных реализаций LSTM.

Дело в том, что вся «обвязка» LSTM в виде гейтов по большому счету служит для того, чтобы управлять состоянием ячейки памяти $\boldsymbol{c}$. Однако если вы посмотрите на формулы, определяющие стандартный LSTM, вы увидите, что непосредственно $\boldsymbol{c}$ для этих гейтов недоступно! Все, что они «видят», — это предыдущее значение $\boldsymbol{h}_{t-1}$, а оно получается из $\boldsymbol{c}_{t-1}$ через нелинейность (что само по себе было бы не страшно) и выходной гейт $\boldsymbol{o}_{t-1}$:

$$\boldsymbol{h}_{t-1} = \boldsymbol{o}_{t-1} \odot \tanh(\boldsymbol{c}_{t-1}).$$

Таким образом, если выходной гейт «закрыт», то есть значения вектора $\boldsymbol{o}$ близки к нулю, получается, что поведение гейтов вообще перестает зависеть от состояния памяти $\boldsymbol{c}$!

Это нежелательный эффект: конечно, мы хотели бы разумно управлять памятью и тогда, когда ячейка не отдает свое значение наружу. Поэтому одна из самых эффективных и потому популярных модификаций LSTM состоит в том, чтобы добавить к базовой конструкции так называемые *замочные скважины* (peepholes), дополнительные соединения, которые позволяют гейтам «подглядывать» текущее значение ячейки памяти $\boldsymbol{c}$:

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma\left(W_{xi}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1} + W_{pi}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i\right) \\
\boldsymbol{f}_t &= \sigma\left(W_{xf}\boldsymbol{x}_t + W_{hf}\boldsymbol{h}_{t-1} + W_{pf}\boldsymbol{c}_{t-1} + \boldsymbol{b}_f\right) \\
\boldsymbol{o}_t &= \sigma\left(W_{xo}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1} + W_{po}\boldsymbol{c}_{t-1} + \boldsymbol{b}_o\right)
\end{aligned}$$

Это, конечно, усложняет конструкцию и добавляет матрицы весов W_{pi}, W_{pf} и W_{po}, но обычно оказывается, что это усложнение того стоит. Мы показываем новые связи в схеме LSTM-ячейки на рис. 6.7, *б*.

Как видите, в конструкции LSTM есть масса разных элементов, служащих для разных целей. Поэтому неудивительно, что существует много разных вариантов LSTM: если начать пытаться пробовать все возможные варианты и осознанно выбирать, какие части архитектуры LSTM нужны вам для конкретной задачи, глаза начинают разбегаться не меньше, чем при выборе функции активации нейронов. В качестве большого исследования, которое покрывает множество разных вариантов, можем порекомендовать работу [336], в которой базовая архитектура LSTM сравнивается с модификациями, каждая из которых так или иначе появлялась в исследованиях:

- LSTM без входного гейта $\boldsymbol{i}_t$;
- LSTM без забывающего гейта $\boldsymbol{f}_t$;
- LSTM без выходного гейта $\boldsymbol{o}_t$;

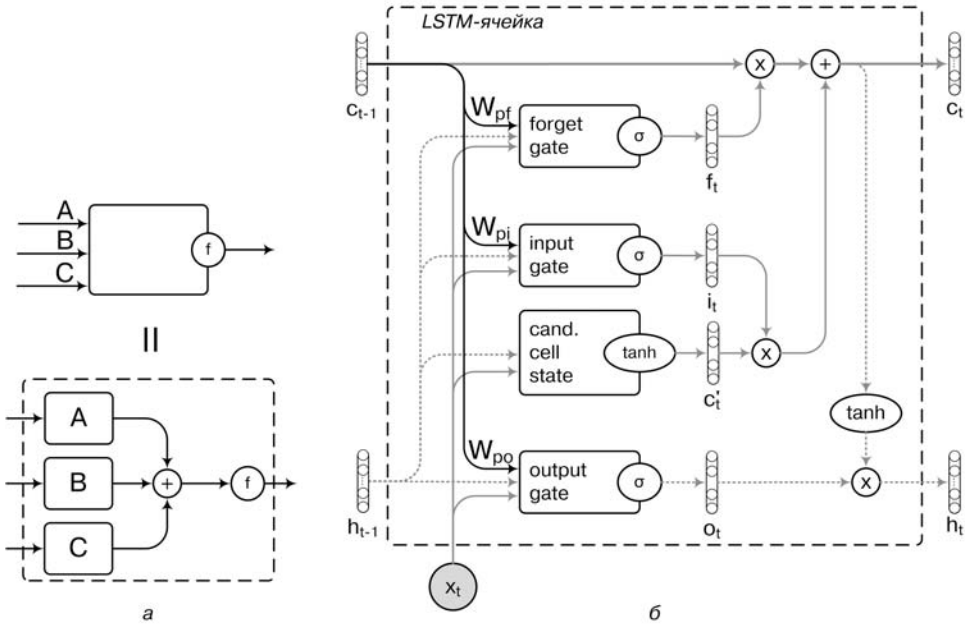


Рис. 6.7. LSTM с замочными скважинами: *а* — обозначение гейта с тремя входами; *б* — структура LSTM-ячейки с замочными скважинами

- LSTM без функции активации σ на входном гейте, то есть:

$$\boldsymbol{i}_t = W_{xi}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1} + W_{pi}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i;$$

- LSTM без функции активации σ на выходном гейте, то есть:

$$\boldsymbol{o}_t = W_{xo}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1} + W_{po}\boldsymbol{c}_{t-1} + \boldsymbol{b}_o;$$

- LSTM без замочных скважин;
- LSTM со связанными входным и забывающим гейтом: вместо того чтобы рассматривать $\boldsymbol{i}_t$ и $\boldsymbol{f}_t$ по отдельности, мы считаем их единым гейтом (назовем его $\boldsymbol{f}_t$), выход которого используется для вычисления $\boldsymbol{c}_t$ как коэффициент выпуклой линейной комбинации:

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + (1 - \boldsymbol{f}_t) \odot \boldsymbol{c}'_t;$$

- LSTM с дополнительными рекуррентными связями на каждом гейте; в этом варианте каждый гейт, кроме своих обычных входов $\boldsymbol{x}_t$ и $\boldsymbol{h}_{t-1}$, получает на вход еще и предыдущие значения всех остальных гейтов:

$$
\begin{aligned}
\boldsymbol{i}_t &= \sigma\left(W_{xi}\boldsymbol{x}_t + W_{hi}\boldsymbol{h}_{t-1} + W_{pi}\boldsymbol{c}_{t-1} + W_{ii}\boldsymbol{x}_{t-1} + W_{fi}\boldsymbol{f}_{t-1} + W_{oi}\boldsymbol{o}_{t-1}\boldsymbol{b}_i\right),\\
\boldsymbol{f}_t &= \sigma\left(W_{xf}\boldsymbol{x}_t + W_{hf}\boldsymbol{h}_{t-1} + W_{pf}\boldsymbol{c}_{t-1} + W_{if}\boldsymbol{x}_{t-1} + W_{ff}\boldsymbol{f}_{t-1} + W_{of}\boldsymbol{o}_{t-1} + \boldsymbol{b}_f\right),\\
\boldsymbol{o}_t &= \sigma\left(W_{xo}\boldsymbol{x}_t + W_{ho}\boldsymbol{h}_{t-1} + W_{po}\boldsymbol{c}_{t-1} + \boldsymbol{b}_o + W_{io}\boldsymbol{x}_{t-1} + W_{fo}\boldsymbol{f}_{t-1} + W_{oo}\boldsymbol{o}_{t-1}\right).
\end{aligned}
$$

Последний вариант добавляет сразу девять новых матриц весов и сильно усложняет конструкцию LSTM. Но, может быть, она от этого станет гораздо более выразительной?

Результаты большого экспериментального исследования в [336] были достаточно ожидаемы: ни один из восьми вариантов LSTM не работал заметно лучше обычного, «ванильного» LSTM. Выяснилось, что ключевыми компонентами LSTM являются забывающий гейт и функция активации на выходном гейте: без забывающего гейта качество сразу сильно проседает, а без функции активации на выходе этот самый выход теоретически может расти неограниченно, что приводит к нежелательным эффектам.

Однако важным и интересным результатом стало то, что некоторые из этих вариантов существенно проще базового LSTM, а работают практически ничем не хуже! В частности, усложненный вариант с дополнительными рекуррентными связями работает даже чуть хуже базового, так что его использовать точно не надо. Вариант без «замочных скважин» практически не уступил варианту с ними. А самым интересным результатом оказалось то, что LSTM со связанными входным и забывающим гейтом, в котором фактически на один гейт меньше, ничем не уступил базовому LSTM.

Все это приводит к мысли о том, что можно попытаться разработать архитектуру, которая сохраняла бы положительные качества LSTM, но при этом была бы заметно проще. Такую архитектуру мы сейчас и рассмотрим.

6.4. GRU и другие варианты

> — Закон у нас простой: вход — рубль, выход — два. Это означает, что вступить в организацию трудно, но выйти из нее — труднее. Теоретически для всех членов организации предусмотрен только один выход из нее — через трубу.
>
> *В. Суворов. Аквариум*

Как мы только что обсуждали, архитектура LSTM требует довольно значительных ресурсов. В обычном RNN каждая ячейка имела один вектор скрытого состояния $\boldsymbol{h}$, а веса были представлены тремя матрицами (плюс свободные члены): матрица весов для входов U, матрица рекуррентных весов W и матрица весов для выходов V. В LSTM-ячейке весов становится гораздо больше. Даже в базовой модели участвует сразу *восемь* матриц весов: $W_{xc}, W_{xi}, W_{xf}, W_{xo}, W_{hc}, W_{hi}, W_{hf}, W_{ho}$. Если

добавить замочные скважины, появятся еще три матрицы: W_{pi}, W_{pf} и W_{po}; хорошо хоть оказалось, что полные рекуррентные связи между всеми гейтами добавлять не нужно. А при обучении каждый из слоев в блоке при раворачивании сети оказывается скопирован столько же раз, сколько элементов имеется в каждой последовательности, так что у больших сетей, составленных из LSTM, будет действительно очень много параметров.

Можно ли добиться того же эффекта долгосрочной памяти и решить проблему затухающих градиентов более эффективно? Эксперименты в [336], которыми мы закончили предыдущий параграф, намекают, что это действительно возможно: оказывается, что критически важными компонентами для успешной работы LSTM выступают по сути только два гейта: выходной и забывающий. Кроме того, понятно, что ключевым моментом является сама «память» $\boldsymbol{c}_t$ и карусель константной ошибки, которая позволяет состоянию LSTM сохраняться надолго. Все остальное можно пытаться как-то сокращать.

Практика последних лет показала, что из перечисленных выше наиболее перспективным оказывается вариант LSTM со связанными входным и забывающим гейтом; от него уже буквально один шаг до упрощенной модели, которая в последнее время устойчиво набирает популярность и постепенно вытесняет LSTM во многих реальных приложениях. Эта модель получила название *gated recurrent unit*, сокращенно GRU; неприятная для русского уха получилась аббревиатура, но что поделать. GRU появились в 2014 году, в работе [399]; см. также эксперименты в [140]. В этой архитектуре используется как раз идея совмещения выходного и забывающего гейта, а скрытое состояние $\boldsymbol{h}_t$ совмещено со значением памяти $\boldsymbol{c}_t$.

Начнем с формул. Вот как работает одна GRU-ячейка:

$$\begin{aligned}
\boldsymbol{u}_t &= \sigma(W_{xu}\boldsymbol{x}_t + W_{hu}\boldsymbol{h}_{t-1} + \boldsymbol{b}_u),\\
\boldsymbol{r}_t &= \sigma(W_{xr}\boldsymbol{x}_t + W_{hr}\boldsymbol{h}_{t-1} + \boldsymbol{b}_r),\\
\boldsymbol{h}'_t &= \tanh(W_{xh'}\boldsymbol{x}_t + W_{hh'}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1})),\\
\boldsymbol{h}_t &= (1 - \boldsymbol{u}_t) \odot \boldsymbol{h}'_t + \boldsymbol{u}_t \odot \boldsymbol{h}_{t-1}.
\end{aligned}$$

Здесь $\boldsymbol{u}_t$ — это гейт обновления (update gate), который и является комбинацией входного и забывающего гейтов. А $\boldsymbol{r}_t$ — это гейт перезагрузки (reset gate); он тоже отвечает за то, какую часть памяти нужно перенести дальше с прошлого шага, но делает это еще до применения нелинейной функции. Ячейка памяти и выход блока $\boldsymbol{h}_t$ тут, в отличие от LSTM, никак не разделяются, и следующий выход $\boldsymbol{h}_t$ получается как комбинация (задаваемая гейтом $\boldsymbol{u}_t$) предыдущего выхода $\boldsymbol{h}_{t-1}$ и текущего кандидата в выход $\boldsymbol{h}'_t$, который, в свою очередь, тоже зависит от $\boldsymbol{h}_{t-1}$, но на этот раз через гейт перезагрузки $\boldsymbol{r}_t$.

Все это мы проиллюстрировали на рис. 6.8, *а*, где показана структура графа вычислений для одной GRU-ячейки. Обратите внимание, что гейтов и матриц весов стало меньше.

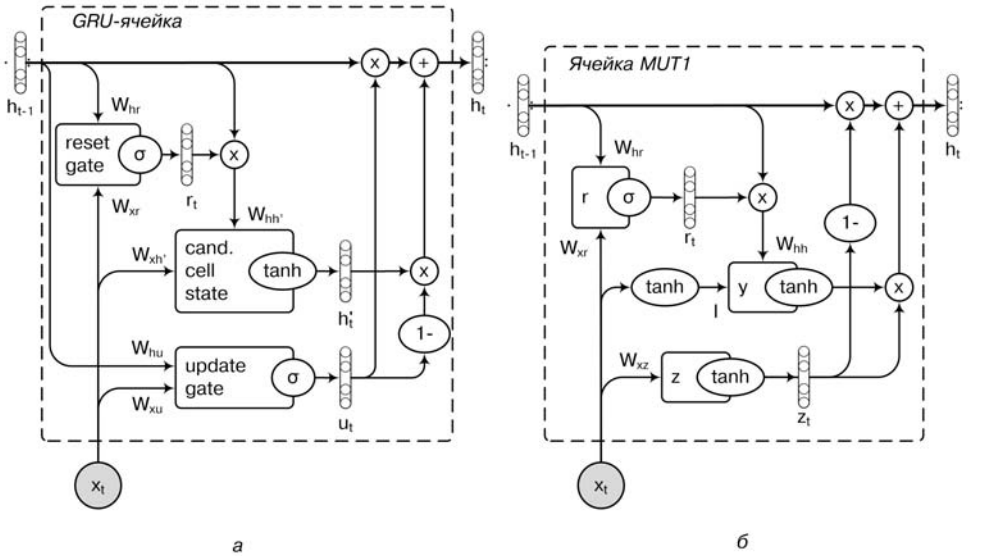


Рис. 6.8. Варианты рекуррентных ячеек:
а — структура ячейки GRU; *б* — структура ячейки MUT1 [267]

Интуиция здесь в том, что t определяет, как объединить новый вход с имеющейся памятью, а $\boldsymbol{u}_t$ — какую часть имеющейся памяти оставить неизменной. Обычную RNN опять можно получить как частный случай GRU, если установить все компоненты t в единицу (всегда брать предыдущее состояние $\boldsymbol{h}_{t-1}$ целиком для вычисления вектора $\boldsymbol{h}'_t$), а все компоненты $\boldsymbol{u}_t$ в ноль (целиком заменять память новым кандидатом $\boldsymbol{h}'_t$). С точки же зрения градиентов мы снова видим, что между $\boldsymbol{h}_t$ и $\boldsymbol{h}_{t-1}$ есть линейная связь, карусель константной ошибки работает, и проблема затухающих градиентов успешно решается.

Основная разница между GRU и LSTM состоит в том, что GRU пытается сделать двумя гейтами то же самое, что LSTM делает тремя. Обязанности забывающего гейта $\boldsymbol{f}$ в LSTM здесь разделены между двумя гейтами, t и $\boldsymbol{u}_t$. Кроме того, не возникает второй нелинейности на пути от входа к выходу, как в случае LSTM. Заметим еще, что здесь опять нужно правильно проинициализировать свободные члены в гейте обновления $\boldsymbol{u}_t$: свободные члены $\boldsymbol{b}_u$ должны быть большими, иначе опять возникнет нежелательный эффект с экспоненциальным затуханием «памяти» в последовательности GRU.

Практика показывает, что GRU практически всегда работает так же или почти так же хорошо, как LSTM. А параметров у него получается гораздо меньше: шесть матриц весов против восьми в простейшем варианте или одиннадцати в более часто используемом (с замочными скважинами). Из-за значительно меньшего числа параметров тренировать GRU проще, чем LSTM, поэтому в некоторых задачах

GRU показывает результаты лучше. Более того, можно позволить себе уместить в той же памяти и с теми же ресурсами сети из большего числа GRU-ячеек, чем LSTM. Поэтому GRU часто используются вместо классических LSTM.

GRU — очень популярный вариант модификации LSTM, но, конечно, не единственный. Из недавних работ отметим [385], где LSTM модифицируется так, чтобы лучше работать с непрерывными потоками событий. Например, беспилотный автомобиль должен быть снабжен огромным количеством самых разных датчиков: видеокамеры следят за дорогой, гироскопы за устойчивостью, особые датчики выдают информацию об уровне топлива и масла, и т. д. и т. п. Все они работают с совершенно разной частотой: важные события приходят с камер гораздо чаще, чем от датчика топлива. Тем не менее, если мы будем обрабатывать эти входы одной и той же рекуррентной сетью, нам придется обрабатывать время дискретно, шаг за шагом, и выбрать одну и ту же частоту дискретизации.

Поэтому в [385] предлагается добавить в LSTM еще один гейт, *гейт времени* (time gate) k_t, который открывается и закрывается периодически в зависимости от трех параметров: параметр τ определяет период (в виде обычного вещественного числа), r_{on} определяет, какую долю времени этот гейт будет открыт, а параметр s — фазовый сдвиг каждого гейта в рамках периода τ.

Предложенная в [385] конструкция гейта времени постепенно «открывается» в течение первой половины своего времени r_{on}, потом постепенно «закрывается» обратно, а в закрытом состоянии немножко «протекает» по аналогии с протекающим ReLU.

Формально это проще всего записать через вспомогательную переменную ϕ_t, показывающую, в какой части своего периода находится гейт времени:

$$\phi_t = \frac{(t-s) \mod \tau}{\tau}, \quad k_t = \begin{cases} \frac{2\phi_t}{r_{\text{on}}}, & \text{если } \phi_t < \frac{1}{2} r_{\text{on}}, \\ 2 - \frac{2\phi_t}{r_{\text{on}}}, & \text{если } \frac{1}{2} r_{\text{on}} < \phi_t < r_{\text{on}}, \\ \alpha\phi_t, & \text{в остальных случаях.} \end{cases}$$

Оказывается, что такая модификация, получившая название *фазированный LSTM* (Phased LSTM), добавляет рекуррентным сетям немало полезных свойств. Фактически Phased LSTM работает одновременно и как преобразование Фурье с настраиваемыми параметрами, что позволяет различать частоты входов, и как своеобразная форма дропаута, зависящего от времени. В результате сети из фазированных LSTM обучаются гораздо быстрее, чем из обычных, и часто превосходят их в точности; несложно придумать и аналогичный вариант GRU.

И в заключение отметим еще одну работу. На протяжении двух последних параграфов мы рассмотрели целый ряд разных архитектур (несколько вариантов LSTM и GRU); все они решают по сути одну и ту же задачу — проблему затухающих градиентов. Но эти архитектуры были придуманы людьми, можно сказать, «из головы», и никто не гарантирует, что это хоть в каком-то смысле «оптимальные» архитектуры, решающие эту задачу.

Поэтому естественным следующим шагом был подход, предложенный в статье [267]; там авторы провели автоматическое исследование более чем десяти тысяч разных архитектур ячеек рекуррентных нейронных сетей, главным образом автоматически порожденных. Они сравнивают эти архитектуры на реальных данных, перебирая множество разных значений гиперпараметров (видимо, здесь очень кстати пришлись вычислительные ресурсы Google, где была сделана эта работа), и в конечном счете пытаются найти наилучшую архитектуру с точки зрения просто практических результатов на классических наборах данных. В результате в работе [267] были найдены три новые архитектуры, чем-то, естественно, напоминающие LSTM и GRU, но показывающие более высокие результаты в ряде экспериментов. Одну из таких новых рекуррентных ячеек, которая в [267] выступает под кодовым названием MUT1, мы для примера показали на рис. 6.8, *б*. Стоит ли этими новыми ячейками пользоваться на практике — вопрос пока открытый, но почему бы при случае и не попробовать?

6.5. SCRN и другие: долгая память в обычных RNN

Особенно в узком кругу.

Б. Гребенщиков

Мы уже не раз говорили и повторим еще раз, что главная содержательная проблема рекуррентных сетей — это как добавить в них как можно более долгосрочную память. В обычных рекуррентных сетях влияние состояний затухает экспоненциально. До сих пор мы рассматривали решения, которые добиваются нужного эффекта, но существенно усложняют модель: в LSTM и GRU долгосрочную память удается смоделировать явным образом, но это стоит недешево, и обучать рекуррентные сети из ячеек LSTM и GRU дольше и сложнее. В этом разделе мы поговорим о более глубоких причинах сложностей, возникающих с обучением рекуррентных сетей, более формально объясним, почему, собственно, градиенты гораздо чаще взрываются или затухают в рекуррентных архитектурах, и попробуем сделать из этого практические выводы о том, что нужно делать.

Этот раздел будет чуть более техническим, чем другие, и его результаты дальше использоваться не будут, так что при ознакомительном чтении книги его можно пропустить. Многое из того, о чем мы говорим в этом разделе, основано на работе [412] и на конструкции, недавно предложенной Томашем Миколовым и соавторами из Facebook в работе [305]. Данные подходы дают эффект долгой памяти в рекуррентной сети, похожий на эффект карусели константной ошибки, но при этом ограничиваются «обычными» классическими нейронами. Хотя эти модели и методы их обучения еще не стали полностью общепринятыми, нам кажется, что они или их модификации имеют на это все шансы.

Начнем с уже знакомой нам классической архитектуры, которую в [305] называют SRN (Simple Recurrent Net):

$$\boldsymbol{s}_t = f(U\boldsymbol{x}_t + W\boldsymbol{s}_{t-1} + \boldsymbol{b}), \quad \boldsymbol{y}_t = h(U\boldsymbol{s}_t + \boldsymbol{c}),$$

где $\boldsymbol{x}_t$ — вход во время t размерности d, $\boldsymbol{s}_t$ — скрытое состояние сети размерности m, $\boldsymbol{y}_t$ — выход во время t размерности d, $U - d \times m$ матрица преобразования входа, $W - m \times m$ матрица рекуррентных весов, $V - m \times d$ матрица преобразования выходов, а f и h — нелинейности, например $f = \tanh$, $h = \text{softmax}$.

Проблемы такой архитектуры нам известны. С одной стороны, градиенты взрываются, но с этим можно отчасти справиться искусственным обрезанием или ренормализацией градиентов. С другой стороны, градиенты обычно быстро затухают, что не позволяет хранить долгую память. В классических рекуррентных моделях (не LSTM или GRU) исследователи долгое время полагали, что это просто проблема градиентного спуска в целом. Однако не исключено, что на самом деле это может быть проблемой именно архитектуры простой рекуррентной сети, и мы сможем предложить способ с этой проблемой справиться. Суть проблемы здесь двоякая: во-первых, нелинейность приводит к тому, что градиенты убывают и со временем затухают совсем, а во-вторых, сама полносвязность скрытого слоя, наличие рекуррентных связей между всеми парами нейронов скрытого состояния сети приводит к тому, что такой слой полностью меняет состояние на каждом временном шаге.

Чтобы избавиться от этих недостатков, нам нужно сначала ненадолго погрузиться в историю развития рекуррентных нейронных сетей. Одной из первых их конструкций была архитектура *сети Джордана* (Jordan network; см. рис. 6.9), разработанная во второй половине 1980-х годов Майклом Джорданом[1] [264–266]. Идея сети Джордана состояла в том, что к обычной двухуровневой нейронной сети добавлялся еще один дополнительный слой нейронов, хранящих контекст (в разных работах они назывались state units или context units), которые получают на вход, во-первых, значения нейронов выходного слоя, а во-вторых, собственные значения с предыдущего шага. Нейроны контекста предоставляли сети теоретическую возможность запоминать и переносить свое состояние с предыдущего шага.

Вскоре, в конце 1980-х, Джеффри Элман упростил сеть Джордана до новой архитектуры, известной как *сеть Элмана* (Elman network; см. рис. 6.10) [139]. Отличие сети Элмана состоит в том, что нейроны контекста теперь получают на вход не выходы исходной сети, а значения активации нейронов скрытого уровня. В своих работах Элман показал, что его сеть способна обучить достаточно долгосрочные временные зависимости. И это при том, что обучение в тех работах было ограничено одним шагом назад по времени, а веса от нейронов скрытого слоя к нейронам

[1] Нет, не тем Майклом Джорданом, который в то время начинал составлять серьезную конкуренцию Ларри Берду и Мэджику Джонсону.

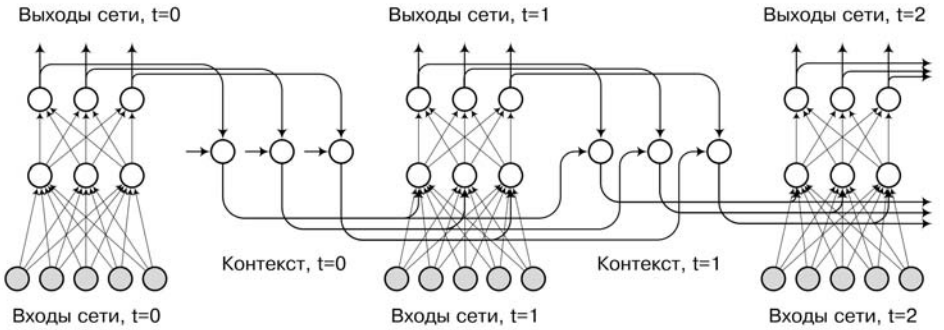


Рис. 6.9. Рекуррентная нейронная сеть Джордана

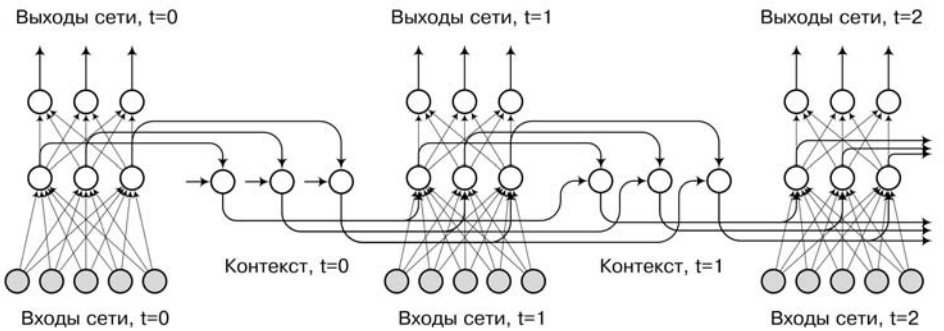


Рис. 6.10. Рекуррентная нейронная сеть Элмана

контекста (как и веса от выходов к нейронам контекста в сети Джордана) не обучались вовсе, а были всегда равны единице! Видимо, сеть Элмана что-то делала правильно — давайте попробуем разобраться...

Чем сеть Элмана отличается от базовой сети SRN? Здесь все очень похоже: тоже есть нейроны, отвечающие за скрытое состояние, и они тоже связаны с предыдущими. Но есть и принципиальная разница: здесь нет никакой нелинейности f между временными шагами, а вместо матрицы рекуррентных весов W в сети Элмана, по сути, используется единичная матрица! Действительно, каждый нейрон контекста связан только с одним, строго соответствующим ему нейроном скрытого слоя, и вес между ними фиксирован и равен единице.

Вся прелесть здесь в том, что правило обновления скрытого состояния, то есть векторов контекста $\boldsymbol{c}_t$, получается таким:

$$\boldsymbol{c}_t = \boldsymbol{c}_{t-1} + U\boldsymbol{x}_t.$$

Здесь буквально по определению $\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = 1$, и карусель константной ошибки выходит сама собой! Конечно, такая модель будет не слишком выразительной — но, может быть, можно каким-то образом соединить выразительность обычной архитектуры и долгосрочную память элмановской?

Основная идея остается той же: вместо полносвязной рекуррентной матрицы, использующей плотную матрицу рекуррентных весов, мы хотим использовать архитектуру, в которой нет никакой нелинейности, а матрица весов диагональная, то есть нейрон скрытого слоя связан со входом, выходом и собой, но не с другими нейронами скрытого слоя. Эта идея, как и многие другие в нейронных сетях, не нова и восходит к работам Майкла Мозера, начатым, опять же, еще в конце 80-х годов прошлого века [377, 378]. При таком подходе очевидно, что градиент матрицы рекуррентных весов никогда не затухнет, что обеспечивает эффект долгосрочной памяти... но одновременно приводит к тому, что и обучать такую модель эффективно не получится: выходит, что на каждом шаге мы должны возвращать градиенты назад до самого начала обучающей последовательности.

Оказывается, что лучше объединить архитектуры. В работе [305] Миколов с соавторами предлагают конструкцию так называемой *структурно ограниченной рекуррентной нейронной сети* (Structurally Constrained Recurrent Network, SCRN). Это рекуррентная сеть с двумя разными скрытыми уровнями, один из которых поддерживает обычное скрытое состояние $\boldsymbol{s}_t$ с полной рекуррентной матрицей W, а другой — медленно меняющийся контекст $\boldsymbol{c}_t$ с диагональной рекуррентной матрицей. Правила вычисления и обновления в подобной сети выглядят так:

$$\begin{aligned}
\boldsymbol{c}_t &= (1-\alpha)\, A\boldsymbol{x}_t + \alpha \boldsymbol{c}_{t-1}, \\
\boldsymbol{s}_t &= f(P\boldsymbol{c}_t + U\boldsymbol{x}_t + W\boldsymbol{s}_{t-1}), \\
\boldsymbol{y}_t &= h(V\boldsymbol{s}_t + B\boldsymbol{s}_t).
\end{aligned}$$

То же самое можно представить и как обычную рекуррентную сеть, объединив $\boldsymbol{s}_t$ и $\boldsymbol{c}_t$ в один вектор. Единственной разницей будет то, что теперь матрица рекуррентных весов M будет не полносвязной, а частично диагональной:

$$M = \begin{pmatrix} R & P \\ \mathbb{O} & \alpha\mathbb{I} \end{pmatrix},$$

где единичная матрица имеет размерность, соответствующую размеру вектора контекста $\boldsymbol{c}_t$.

Мы изобразили эту архитектуру на рис. 6.11: обратите внимание на «магистраль» в верхней части картинки, по которой движется с течением времени вектор контекста $\boldsymbol{c}_t$. Примеры кода SCRN мы в книге приводить не будем, но полная реализация этой модели выложена в общий доступ группой Миколова[1].

[1] http://github.com/facebook/SCRNNs

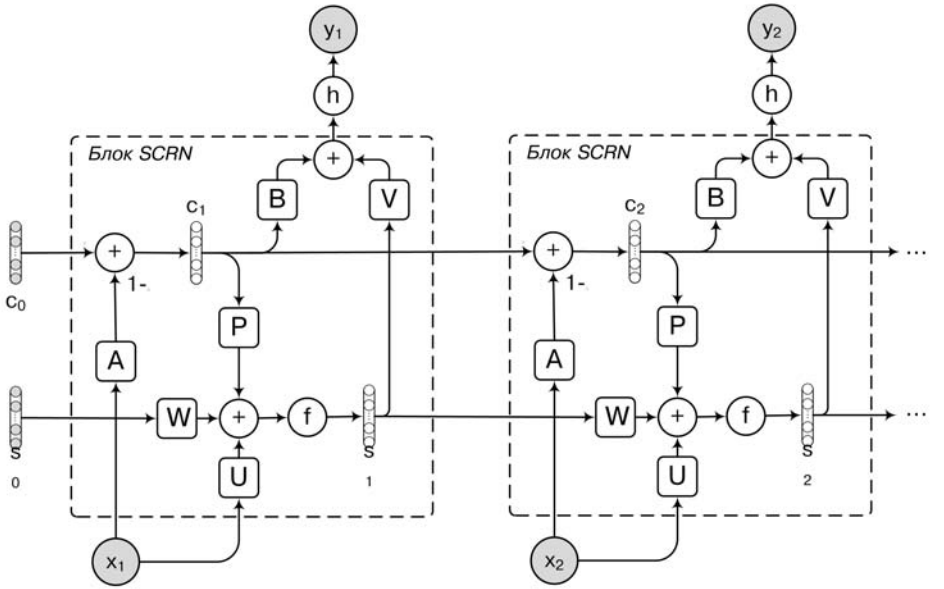


Рис. 6.11. Структурно ограниченная рекуррентная нейронная сеть

В работе [305] приводятся результаты практического сравнения с LSTM, и получается весьма неплохо: фактически SCRN добиваются тех же результатов, что LSTM, с таким же числом рекуррентных ячеек, но параметров-то у них в четыре раза меньше! Поэтому обучаются SCRN в результате быстрее; как конкретный пример архитектуры авторы приводят 100 полносвязных нейронов и 40 нейронов контекста.

На больших наборах данных эффект от такого «кэша» становится еще больше, и SCRN даже начинают побеждать LSTM по качеству. Дело в том, что SCRN накапливает больше истории в «медленном» скрытом уровне; в целом, вывод состоит в том, что SCRN лучше работают для больших данных, и по качеству результата, и потому, что их очень дешево тренировать по сравнению с LSTM.

Есть и альтернативный подход к проблеме. Давайте попробуем сделать матрицу рекуррентных весов, матрицу переходов между скрытыми состояниями унитарной/ортогональной, то есть такой, чтобы ее норма была равна единице, и градиенты не взрывались и не затухали буквально по определению. Несколько недавних работ предлагают разные подходы к тому, как это сделать.

Во-первых, отметим одну очень простую идею, которая, тем не менее, может привести к заметным улучшениям: давайте просто инициализируем рекуррентные веса единичной матрицей! У единичной матрицы, естественно, норма равна единице, и если при этом используется ReLU в качестве функции активации, градиенты

не будут затухать или взрываться по крайней мере поначалу. Работа [300] показывает, что этот простой трюк может значительно улучшить обучение обычных рекуррентных сетей, и они после этого начинают лучше обрабатывать долгосрочные зависимости, добираясь почти что до уровня сетей LSTM.

Во-вторых, можно ввести правильную регуляризацию; этот подход предлагается в [412]. Если матрица W не будет ортогональной/унитарной, то проблема будет выражаться в том, что производная функции ошибки по $\boldsymbol{s}_t$ мультипликативно отличается от производной функции ошибки по $\boldsymbol{s}_{t+1}$, и эта ошибка накапливается. Давайте явным образом попросим, чтобы эти производные отличались не слишком! Для этого можно просто добавить к функции ошибки такой регуляризатор:

$$\Omega = \sum_k \Omega_k = \sum_k \left(\left\| \frac{\frac{\partial E}{\partial \boldsymbol{s}_{k+1}} \frac{\partial \boldsymbol{s}_{k+1}}{\partial \boldsymbol{s}_k}}{\frac{\partial E}{\partial \boldsymbol{s}_{k+1}}} \right\| - 1 \right)^2.$$

В числителе дроби здесь стоит производная сложной функции, по сути $\frac{\partial E}{\partial \boldsymbol{s}_k}$. Иначе говоря, вводя этот регуляризатор, мы хотим, чтобы отношение производных $\frac{\partial E}{\partial \boldsymbol{s}_k}$ и $\frac{\partial E}{\partial \boldsymbol{s}_{k+1}}$ было близко к единице. Мы не вводим явных ограничений на матрицу W, а мягко, но настойчиво рекомендуем ее норме быть поближе к единице.

Но можно попробовать и более жесткий подход. В работе [9] предлагаются так называемые *унитарные рекуррентные сети* (unitary RNN, uRNN), в которых матрица рекуррентных весов всегда строго унитарная. Как сделать матрицу унитарной? Для этого придется «сконструировать» ее из параметрически заданных блоков, каждый из которых унитарен (при умножении матриц унитарность сохраняется). В [9] предлагается искать такую матрицу в виде произведения

$$W = D_3 R_2 F^{-1} D_2 \Pi R_1 F D_1,$$

где D — диагональные матрицы, F — преобразование Фурье, R — матрица отражения, а Π — перестановочная матрица; все это подобрано так, чтобы в произведении всегда получалась унитарная матрица. Интересно, что весов суммарно становится даже меньше: мы сильно ограничили множество всех возможных матриц, так что у этой модели $O(n)$ параметров вместо $O(n^2)$. Кстати, все матрицы здесь комплексные, комплексными получаются скрытые состояния, а в качестве функции активации предлагается использовать комплексный вариант ReLU:

$$\text{modReLU}(z) = \begin{cases} (|z| + b)\frac{z}{|z|} & \text{при} \quad |z| + b \geq 0, \\ 0 & \text{при} \quad |z| + b < 0. \end{cases}$$

Теперь нет ни взрывающихся, ни затухающих градиентов: унитарная матрица взорваться не может. Работа [9] утверждает, что этот подход гораздо лучше работает с длинными зависимостями, чем LSTM, «запоминая» до тысячи шагов.

А в совсем свежей работе [318] строятся конструкции рекуррентных гейтов SRU (Simple Recurrent Unit), которые значительно ускоряют вывод по сравнению со стандартными LSTM и GRU. Идея состоит в том, что в обычных рекуррентных сетях следующее скрытое состояние $\boldsymbol{h}_t$ зависит от предыдущего $\boldsymbol{h}_{t-1}$, что не позволяет вычислять $\boldsymbol{h}_t$ параллельно. А в SRU такой зависимости просто нет: состояние ячейки памяти $\boldsymbol{c}_t$ используется и «протаскивается» через время, но $\boldsymbol{h}_{t-1}$ в рекурсии не участвует. В результате значения почти всех гейтов становятся независимыми и могут вычисляться параллельно. Авторы показывают, что такой подход приводит к существенному ускорению (в несколько раз) без потери качества; впрочем, эта идея тоже еще не подтверждена обширной практикой.

Подведем итог. Кажется, что относительно простые трюки имеют шансы преодолеть проблемы взрывающихся и затухающих градиентов, которые так долго останавливали прогресс в этой области. Однако это пока количественный прогресс, а не качественный: мы пытаемся сделать «длинную память» на тысячу шагов, но просто взять и положить куда-нибудь значение, чтобы потом его вытащить, когда нужно — хоть через миллион шагов, хоть через миллиард, — такими способами не получается. Поэтому рекуррентные нейронные сети пока не справляются даже с довольно простыми задачами: например, обучить и распознать какое-нибудь простое правило порождения строчек, скажем, распознавать строчки вида $a^n b^n$ (последовательность букв a, за которой идет последовательность из такого же числа букв b). Здесь еще остается большой простор для дальнейших исследований.

6.6. Пример: порождаем текст символ за символом

> Несмотря на эти три препятствия, фрагментарный Дон Кихот Менара — вещь более тонкая, чем роман Сервантеса. Последний довольно прямолинейно противопоставляет причудливым рыцарским домыслам бедную провинциальную действительность своей страны; Менар избирает в качестве «действительности» родину Кармен времен Лепанто и Лопе.
>
> *Х. Л. Борхес. Пьер Менар, автор «Дон Кихота»*

В этом разделе мы дадим небольшой, но очень интересный пример, связанный с задачей так называемого *языкового моделирования* (language modeling), в которой целевая функция состоит в том, чтобы предсказывать, как данный текст будет продолжаться дальше. В главе 7 мы обсудим основные понятия интеллектуальной обработки текстов, дадим краткий обзор ключевых методов, которыми обрабатывали тексты до революции машинного обучения и будем подробно и планомерно рассматривать современные подходы к обработке текстов, основанные на глубоких нейронных сетях. И к задаче языкового моделирования тоже еще обязательно вернемся. А сейчас мы просто хотим продемонстрировать способности современных

рекуррентных архитектур, поэтому рассмотрим не слишком подходящую для серьезных применений к естественному языку, но крайне интересную модель, основанную на работах [192, 273] и документации к библиотеке Keras [79]. Кроме того, этот пример позволит нам познакомиться с несколькими важными и часто использующимися на практике возможностями библиотеки Keras; так что даже если вы не интересуетесь порождением текстов, пропускать этот пример не рекомендуем.

Идея очень проста: давайте будем пытаться порождать текст *буква за буквой*, рассматривая его просто как поток символов (то есть дискретных объектов). Тогда задача языкового моделирования очень легко формализуется: мы хотели бы предсказывать следующий символ по предыдущему тексту. Решать ее мы будем рекуррентной нейронной сетью, которая на вход должна получить последовательность символов, то есть просто много-много текста, а на выходе предсказать следующий символ, то есть решить задачу классификации на несколько десятков классов.

Как определить тренировочные данные для такого посимвольного порождения текста? Некоторая проблема состоит в том, что даже рекуррентная сеть на практике не может работать с неограниченной последовательностью, мы должны както «нарезать» ее на тренировочные примеры и мини-батчи. Можно предложить несколько вариантов того, как это сделать. Первый, самый простой вариант — взять и разбить текст на последовательности фиксированной длины (окна) и предсказывать следующий по предыдущим. Это будет так или иначе работать, но получится, что начало и конец тренировочного примера будут достаточно «внезапными», и артефакты по краям будут мешать генерировать хороший текст на выходе.

Второй вариант, которым мы и воспользуемся, таков: мы будем дробить текст на последовательности разной длины, но не слишком большой, например на предложения. Затем мы добавим спецсимволы начала и конца предложения и дополним каждую такую строчку до максимума еще одним новым спецсимволом, чтобы уравнять длины предложений. Тогда сеть должна будет обучиться генерировать нормальные законченные предложения, в том числе показывать, когда очередное предложение закончится.

Есть и третий, пожалуй, самый лучший вариант для этой задачи: нарезать текст на последовательности примерно одной длины, но при этом правильным образом инициализировать скрытые состояния рекуррентной сети. Для этого нужно взять наш длинный-длинный вход и превратить его сначала в достаточно «широкий» прямоугольник размера $N \times L$, где L — длина каждого тренировочного примера, а N — число тренировочных примеров в исходной последовательности. Пока что L слишком велика, но теперь можно разрезать этот прямоугольник на более короткие мини-батчи, так сказать, «по вертикали»; получается, что второй батч — это продолжение первого и его нужно обучать не с нуля, а с того скрытого состояния, на котором закончился предыдущий батч. Это можно сделать в стандартных библиотеках для глубокого обучения: перенести скрытые состояния рекуррентной сети из конца предыдущего мини-батча в начало следующего. Но сам предыдущий мини-батч при этом будет все-таки забыт, и градиент будет проходить назад только

по коротким подпоследовательностям, до начала текущего мини-батча. Если вам понадобится действительно хорошо предсказывать последовательности, мы рекомендуем использовать именно этот подход, но он требует довольно много технической работы, поэтому в примере кода ниже мы просто будем делить текст на предложения.

Итак, давайте начнем смотреть код. Прежде всего нужно прочитать входной файл (получающимся у нас сетям бессмысленно будет подавать на вход гигантские корпусы, поэтому мы не будем беспокоиться о том, что прочитаем входной файл два раза) и составить список символов. В коде ниже `input_fname` — это имя входного файла.

```
START_CHAR = '\b'
END_CHAR = '\t'
PADDING_CHAR = '\a'
chars = set( [START_CHAR, '\n', END_CHAR] )
with open(input_fname) as f:
  for line in f:
    chars.update( list(line.strip().lower()) )
char_indices = { c : i for i,c in enumerate(sorted(list(chars))) }
char_indices[PADDING_CHAR] = 0
indices_to_chars = { i : c for c,i in char_indices.items() }
num_chars = len(chars)
```

Обратите внимание, что мы определили три спецсимвола: `START_CHAR` будет подставляться перед началом предложения, `END_CHAR` после его конца, а `PADDING_CHAR` будет заполнять остаток предложения до максимума длины. Для простоты мы также не будем различать строчные и заглавные буквы, а просто пропустим каждую строчку через `lower()`.

Дальше создадим векторные представления для символов; это будет просто one-hot представление, в котором каждый символ представляется вектором с одной единицей, за исключением спецсимвола `PADDING_CHAR`, который разумно представлять просто нулевым вектором.

```
def get_one(i, sz):
    res = np.zeros(sz)
    res[i] = 1
    return res

char_vectors = {
    c : (np.zeros(num_chars) if c == PADDING_CHAR else get_one(v, num_chars))
    for c,v in char_indices.items()
}
```

Дальше прочитаем входной файл еще раз, теперь уже деля его на предложения и выписывая их отдельно. Будем брать только предложения длиннее 10 символов.

```
sentence_end_markers = set( '.!?' )
```

```
sentences = []
current_sentence = ''
with open( input_fname, 'r' ) as f:
    for line in f:
        s = line.strip().lower()
        if len(s) > 0:
            current_sentence += s + '\n'
        if len(s) == 0 or s[-1] in sentence_end_markers:
            current_sentence = current_sentence.strip()
            if len(current_sentence) > 10:
                sentences.append(current_sentence)
            current_sentence = ''
```

Следующий шаг — векторизация. Давайте определим процедуру, которая превращает набор предложений в два тензора: `X` содержит векторы символов, а `y` — результат, который нам нужно предсказать; это на самом деле ровно тот же тензор `X`, только смещенный на один вектор влево: во время t мы предсказываем символ, который будет стоять на месте $t + 1$.

```
def get_matrices(sentences):
    max_sentence_len = np.max([ len(x) for x in sentences ])
    X = np.zeros((len(sentences), max_sentence_len, len(chars)), dtype=np.bool)
    y = np.zeros((len(sentences), max_sentence_len, len(chars)), dtype=np.bool)
    for i, sentence in enumerate(sentences):
        char_seq = (START_CHAR + sentence + END_CHAR).ljust(
                        max_sentence_len+1, PADDING_CHAR)
        for t in range(max_sentence_len):
            X[i, t, :] = char_vectors[char_seq[t]]
            y[i, t, :] = char_vectors[char_seq[t+1]]
    return X,y
```

В этом коде стоит обратить внимание на две вещи. Во-первых, для ускорения и экономии памяти мы добавили параметр `dtype=np.bool`, превратив все матрицы в булевские: они и так содержали только значения 0 и 1, так что эта экономия ничего не испортит. Во-вторых, мы добавляем в начало каждого предложения `START_CHAR`, в конец `END_CHAR`, а затем дополняем его до `max_sentence_len+1` символом `PADDING_CHAR`; в Python это можно сделать стандартной функцией `ljust`.

Теперь все готово для того, чтобы строить модель и обучать ее. Начнем с самой простой модели: один уровень LSTM-ячеек, результаты которых пропускаются через один полносвязный слой, и тут же происходит классификация. Вот как это выглядит в Keras:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM, TimeDistributed
model = Sequential()
model.add(LSTM(output_dim=128, activation='tanh',
    return_sequences=True, input_dim=num_chars))
```

```
model.add(Dropout(0.2))
model.add(TimeDistributed(Dense(output_dim=num_chars)))
model.add(Activation('softmax'))
```

В этом коротеньком отрывке появились сразу две важных и новых для нас сущности. Первая из них — слой `LSTM`, который мы, как обычно, импортировали из `keras.layers`. Давайте разберем его аргументы:

- параметр `output_dim=128` означает, что наш слой состоит из 128 LSTM-ячеек; соответственно, размерность выхода будет равна 128;
- параметр `activation='tanh'` говорит, что в качестве функций активации в ячейках LSTM мы будем использовать гиперболический тангенс tanh (по умолчанию был бы логистический сигмоид σ); мы не утверждаем, что для этой задачи гиперболический тангенс непременно лучше, но продемонстрировать этот параметр было нужно, так почему бы и не сейчас;
- `input_dim=X.shape[2]`, как и в других слоях, показывает, что на входе ожидаются мини-батчи из векторов длины `X.shape[2]`, то есть (см. выше) длины **`len`**`(chars)`; обратите внимание, что мы не задавали ни размер мини-батча, ни даже длину предложения на входе, все это может остаться переменным;
- и наконец, очень интересный параметр **`return`**`_sequences=True`; дело в том, что по умолчанию слой LSTM будет идти по входной последовательности (в нашем случае — по предложению) от начала до конца, выдавая в качестве результата только свое состояние в конце всего входа, то есть тензор размерности `input_dim` × `output_dim`; а при **`return`**`_sequences=True` слой будет выдавать наверх свои выходы после каждого примера; значит, в нашем случае выходной тензор будет трехмерным, и его размерность будет совпадать с размерностью входного тензора `X`.

Вторая новая сущность — это слой `TimeDistributed`, который служит «оберткой» для полносвязного слоя `Dense`, из которого уже должны получаться активации для каждого из символов, то есть каждого из классов в нашей задаче классификации. Смысл слоя `TimeDistributed` в том, чтобы предоставить возможность использовать одни и те же веса по всей длине входной последовательности, то есть чтобы в полносвязном слое веса были общими (shared weights). Формально это значит, что слой `Dense(output_dim=num_chars)` будет получать на вход каждый из выходов сети LSTM и использовать одни и те же веса, чтобы превратить скрытое состояние этой сети LSTM в предсказанную букву.

И небольшое замечание о дропауте. Дропаут в рекуррентных сетях — это дело достаточно тонкое. Вплоть до очень недавних работ, связанных с байесовским подходом к выводу в нейронных сетях [169], считалось, что дропаут на рекуррентных весах делать не нужно, потому что таким образом разрываются долгосрочные связи: трудно запомнить . Такой взгляд подтверждался подробными экспериментами, например, в работе [583]. Поэтому в большинстве классических рекуррентных моделей дропаут делают только между рекуррентными слоями, но не внутри них. Мы подробно поговорим о последних байесовских новостях дропаута в рекуррентных

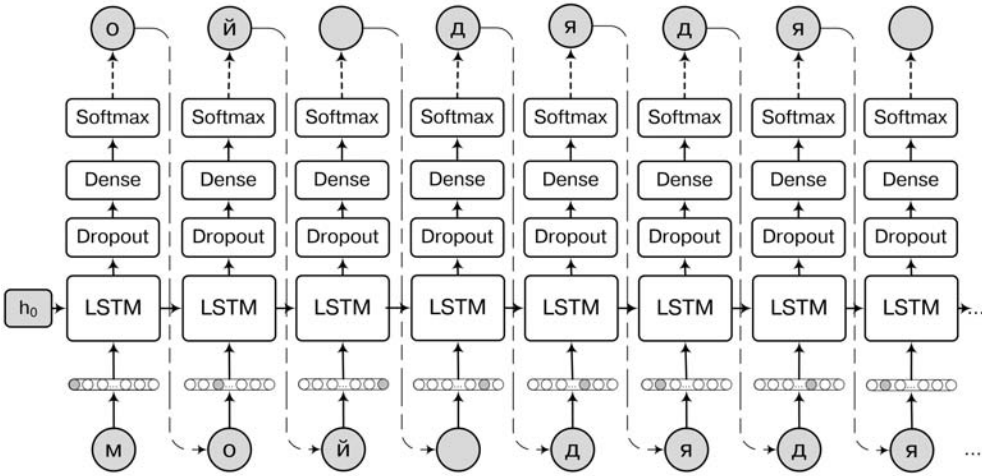


Рис. 6.12. Архитектура нейронной сети для посимвольного порождения текстов на основе одного уровня LSTM-ячеек

сетях в разделе 10.5; там же и объясним, почему раньше думали так и что теперь изменилось. Но в примерах мы сейчас не будем ничего переусложнять (особенно учитывая, что в стандартных библиотеках сейчас, в конце 2016 года, «новый взгляд» на дропаут пока еще не делается в одну строчку) и будем использовать только «стандартный» дропаут между уровнями.

Вся эта модель проиллюстрирована на рис. 6.12: полученные на вход символы сначала преобразуются в векторное (one-hot) представление, затем поступают на вход LSTM-ячейке на очередном шаге времени, затем проходят через дропаут-слой и подаются на вход полносвязному слою, результаты которого уже через softmax порождают собственно результат классификации. А целевой переменной служит следующий символ во входном тексте — это на рис. 6.12 отражено в виде пунктирных линий, связывающих выход и следующий вход.

Теперь скомпилированную модель нужно оптимизировать. Чтобы градиенты в рекуррентной сети не взрывались, их обязательно нужно купировать, обрезать до некоторого максимально допустимого размера (по-английски это называется gradient clipping).

Для этого служат два параметра, которые можно передать любому оптимизатору в Keras:

- `clipnorm` будет масштабировать вектор градиента так, чтобы его норма не превысила заданного порога;
- `clipvalue` будет просто обрезать до заданного порога каждую компоненту градиента по отдельности.

Градиенты всегда нужно купировать при обучении сетей из LSTM или GRU, особенно глубоких; ссылки на это есть во всех статьях, которыми мы пользовались в этом разделе[1]. В нашем случае давайте обрезать градиенты до нормы 1:

```
from keras.optimizers import Adam
model.compile(loss='categorical_crossentropy',
    optimizer=Adam(clipnorm=1.), metrics=['accuracy'])
```

Теперь практически все готово для того, чтобы запускать оптимизацию; но сначала — еще две любопытные детали, которые скорее призваны продемонстрировать удобные возможности библиотеки Keras, чем улучшить обучение модели. Во-первых, мы определили процедуру `get_matrices` так, что размерность тензоров на выходе будет зависеть от максимальной длины предложений на входе. Это сделано специально: теперь мы можем подавать на вход мини-батчи, сгенерированные процедурой `get_matrices`, и в них не обязательно все предложения будут «добиты» нулями до самого длинного во всем тексте, а просто длина предложений в разных мини-батчах будет различаться.

Это удобно, но для этого нужно подавать мини-батчи последовательно на вход процедуре `fit`, а мы пока умеем подавать ей на вход только весь датасет целиком. К счастью, это в Keras сделать несложно: у класса `Model` есть метод `fit_generator`, который принимает на вход не датасет, а функцию-генератор, порождающую мини-батчи один за другим. Давайте такую функцию и напишем. Ниже для простоты мы не стали писать отдельный генератор для валидационного множества, а породили матрицы для него раз и навсегда; 0,05 — это доля валидационного множества.

```
test_indices = np.random.choice(range(len(sentences)), int(len(sentences) * 0.05))
sentences_train = [ sentences[x]
       for x in set(range(len(sentences))) - set(test_indices) ]
sentences_test = [ sentences[x] for x in test_indices]
sentences_train = sorted(sentences_train, key = lambda x : len(x))
X_test, y_test = get_matrices(sentences_test)
batch_size = 16
def generate_batch():
    while True:
        for i in range( int(len(sentences_train) / batch_size) ):
            sentences_batch = sentences_train[ i*batch_size : (i+1)*batch_size ]
            yield get_matrices(sentences_batch)
```

Теперь функция `generate_batch` будет последовательно проходить датасет, разбивая его на мини-батчи по 16 предложений. А чтобы это имело еще больше смысла, мы предварительно отсортировали `sentences_train` по длине, так что теперь

[1] А если в какой-то статье об этом не сказано, то, скорее всего, авторы просто забыли это написать или опустили за очевидностью. Например, в [192] Алекс Грэйвс делает сноску о том, что градиенты обрезались во всех его предыдущих работах и в выложенном им коде, но вплоть до работы [192] он попросту забывал это указывать.

мини-батчи будут иметь последовательно увеличивающуюся ширину, предложения в них будут иметь примерно одинаковую длину, и заполнять PADDING_CHAR придется не так много остатков.

Вторая важная возможность Keras — это функции обратного вызова (callbacks). Метод fit (и fit_generator тоже, конечно же) имеет необязательный параметр callbacks, и если в него передать список функций, то они будут запускаться после каждой эпохи обучения; это очень удобно, потому что не нужно писать собственно цикл обучения. Есть стандартные примеры, которыми можно выписывать лог разных статистик по мере обучения, сохранять веса модели и т. д. Собственно, все графики ошибок и значений функции потерь на тренировочном и валидационном множествах, которые мы все время строили и будем строить в этой книге, делаются именно так. Давайте добавим две стандартные функции обратного вызова:

```
from keras.callbacks import ModelCheckpoint, CSVLogger
cb_sampler = CharSampler(char_vectors, model)
cb_logger = CSVLogger('sin_l/' + model_fname + '.log')
```

Но и это еще не все. Мы сконструировали модель, которая берет на вход тексты и пытается научиться предсказывать следующий символ. Осталось только понять, как теперь, собственно, порождать тексты, и добавить в код что-нибудь, что будет эти тексты выписывать.

На выходе из своего последнего слоя модель порождает веса x_w тех или иных слов w; эти веса не обязательно суммируются в единицу и совершенно не обязательно имеют смысл вероятностей, так что у нас есть выбор, как именно превратить их в вероятности сэмплирования. Функция softmax подсказывает, что получающиеся вероятности разумно выбирать так:

$$p(w) \propto e^{-\frac{1}{T}x_w}.$$

В этой формуле есть свободный параметр T; T называется *температурой* сэмплирования, и слово «температура» здесь используется вполне буквально, в обычном повседневном смысле: в физике температура возникает точно таким же образом. Идея такая: если T большое (температура высокая), то показатели экспоненты будут достаточно небольшими по модулю, результаты возведения в степень тоже будут не слишком большими, и на выходе вероятности будут достаточно близки, то есть сэмплировать мы будем весьма случайно. А если T маленькое (температура низкая), то, наоборот, показатели будут достаточно велики по модулю, и после возведения в степень будут часто получаться очень маленькие числа, практически нули, а ненулевыми окажутся не так много вероятностей; иначе говоря, с маленькой температурой мы гораздо чаще будем получать одно-два значения с самыми большими весами. Доводя до предела, при $T \to \infty$ мы получим равномерное распределение (все показатели равны нулю), а при $T \to 0$ получим распределение, полностью сосредоточенное в исходе с максимальным весом. Давайте увидим этот эффект на численном примере, для конкретного вектора весов $\boldsymbol{x}$:

$$\begin{aligned}
\boldsymbol{x} = [1,2,5], \quad T = 10.0 \quad &\Rightarrow \quad \boldsymbol{p} \approx (0.39, 0.36, 0.25);\\
\boldsymbol{x} = [1,2,5], \quad T = 1.0 \quad &\Rightarrow \quad \boldsymbol{p} \approx (0.72, 0.27, 0.01);\\
\boldsymbol{x} = [1,2,5], \quad T = 0.1 \quad &\Rightarrow \quad \boldsymbol{p} \approx (0.9999, 4 \cdot 10^{-5}, 4 \cdot 10^{-18}).
\end{aligned}$$

В примере получается, что при $T = 10{,}0$ мы сэмплируем практически равномерно, первый исход выигрывает совсем немного; а при $T = 0{,}1$ уже жизни не хватит, чтобы дождаться появления третьего исхода, мы практически всегда будем видеть только первый. При порождении текстов из модели будет тот же эффект: при одних и тех же весах языковой модели сэмплирование с высокой температурой будет более разнообразным, в том числе часто совершенно случайным, а сэмплирование с низкой температурой будет более устойчивым, чаще будет выдавать одно и то же при похожих входах.

Осталось только выписать порождение нескольких текстов в виде еще одной функции обратного вызова. На этот раз нам придется писать ее самостоятельно, поэтому давайте унаследуем ее от класса `keras.callbacks.Callback` и переопределим методы `on_train_begin` (что делать в начале обучения) и `on_epoch_end` (что делать после каждой эпохи).

```
from keras.callbacks import Callback
class CharSampler(Callback):
    def __init__(self, char_vectors, model):
        self.char_vectors = char_vectors
        self.model = model

    def on_train_begin(self, logs={}):
        self.epoch = 0
        if os.path.isfile(output_fname):
            os.remove(output_fname)

    def sample( self, preds, temperature=1.0):
        preds = np.asarray(preds).astype('float64')
        preds = np.log(preds) / temperature
        exp_preds = np.exp(preds)
        preds = exp_preds / np.sum(exp_preds)
        probas = np.random.multinomial(1, preds, 1)
        return np.argmax(probas)

    def sample_one(self, T):
        result = START_CHAR
        while len(result)<500:
            Xsampled = np.zeros( (1, len(result), num_chars) )
            for t,c in enumerate( list( result ) ):
                Xsampled[0,t,:] = self.char_vectors[ c ]
            ysampled = self.model.predict( Xsampled, batch_size=1 )[0,:]
            yv = ysampled[len(result)-1,:]
            selected_char = indices_to_chars[ self.sample( yv, T ) ]
            if selected_char==END_CHAR:
```

```
                break
            result = result + selected_char
        return result

    def on_epoch_end(self, batch, logs={}):
        self.epoch = self.epoch+1
        if self.epoch % 50 == 0:
            print("\nEpoch %d text sampling:" % self.epoch)
            with open( output_fname, 'a' ) as outf:
                outf.write( '\n===== Epoch %d =====\n' % self.epoch )
                for T in [0.3, 0.5, 0.7, 0.9, 1.1]:
                    print('\tsampling, T = %.1f...' % T)
                    for _ in range(5):
                        self.model.reset_states()
                        res = self.sample_one(T)
                        outf.write( '\nT = %.1f\n%s\n' % (T, res[1:]) )
```

Здесь мы использовали вышеописанную процедуру сэмплирования, и теперь будем просить модель генерировать тексты с пятью разными значениями температуры каждые 50 эпох обучения. Все, теперь все готово, можно обучать!

```
model.fit_generator( generate_batch(),
    int(len(sentences_train) / batch_size) * batch_size,
    nb_epoch=1000, verbose=True, validation_data = (X_test, y_test),
    callbacks=[cb_logger, cb_sampler, cb_checkpoint] )
```

Второй аргумент метода `fit_generator` — это число примеров в тренировочной выборке; дело в том, что генератор `generate_batch` должен работать неограниченно долго, читая датасет много раз (чтобы можно было проводить несколько эпох обучения), но при этом процесс обучения должен знать, когда заканчивается очередная эпоха.

Давайте же наконец посмотрим, что получается! Для первого примера мы взяли текст «Евгения Онегина» и подали его на вход, обучив 1000 эпох сети, показанной выше; некоторые из сгенерированных текстов показаны в табл. 6.1. Конечно, осмысленного текста тут ждать не приходится, но результаты довольно интересные. Хотя с точки зрения интерпретации по большому счету мало что изменилось между пятидесятой эпохой и тысячной, видно, что на самом деле тексты стали заметно разнообразнее: после пятидесятой эпохи тексты с $T = 0{,}3$ используют почти исключительно самые частые гласные и согласные, а после тысячной эпохи модель уже понимает, к примеру, корень «люб».

Но давайте продолжать, не будем останавливаться на достигнутом! Следующий этап наших экспериментов — попробовать построить более сложную сеть, которая бы могла обучать и «запоминать» более долгосрочные и более сложные взаимосвязи, и тем самым улучшить результат порождения; здесь мы будем использовать главным образом идеи работы [192].

Таблица 6.1. Примеры порожденных текстов: модель с 1 слоем из 128 LSTM-ячеек, обученная на тексте «Евгения Онегина»

LSTM, эпоха 50, $T = 0{,}3$	*LSTM, эпоха 50,* $T = 0{,}5$	*LSTM, эпоха 50,* $T = 1{,}1$
он что мне под постили страдной, в домно не всем слодом вете, в сомольно продной поровали, и пристить в постали сердце и страдной пораки сласта и в сердце постала не трам, в пором не стриненье нете как верденный свот болена.	так не вожды свеость сместа и в послича танных леном и подражденный вздрегодань, и странывае поластале лестель он как постали слида и ворамо стара седен, заметной сордце сладибам в постила с нем свотой бредной	окknею сжеть х пилапала, педным осетская! друдной, морым ты, прохенные вак; и ыпрамняй зой? глодой поль требетсть гулся, мрудье бреютн, родвобы порожин, подруши, и жирныху таешь; моей входтаков нем нал бостметрой.
LSTM, эпоха 1000, $T = 0{,}3$	*LSTM, эпоха 1000,* $T = 0{,}5$	*LSTM, эпоха 1000,* $T = 1{,}1$
и вы, почтол до душа слаздесь, не свет под ней сторой в тете, не сторо предрежда невень, он он светлиновал не мечь; он в полюбитенный прирад, и старость, и в сердца всей, с пора и мельновой воспода, где весной все в доменый без,	к они в блегной нет оних, и все поволиным в собед, и полногод и свой продтонный, не почальстахом бестриланье, болово того жиним 3, однам волшенный приглоснет и заменили и поэт и страшно льго все другом оне.	очем, сестремя, ейзнова, что бурчисть вак, друголир, я востая утрых живой: везди еще страглю им мой, на полкафбаым лупобкад, и не душла, плопеть ворок маны урыней мучшуй.

Основная идея здесь состоит в том, чтобы добавить новые рекуррентные уровни поверх первого. Можно было бы сделать это совсем наивно: взять и добавить еще один слой вида

```
LSTM(output_dim=128, activation='tanh', return_sequences=True)
```

между первым слоем и полносвязной частью.

Однако будет лучше (внимание: это важный трюк в построении глубоких рекуррентных сетей!), если последующему слою подать на вход не только результат предыдущего, но и собственно входную последовательность; такая конструкция называется *skip-layer connection* (связь с пропуском слоя).

Мы проиллюстрировали это на рис. 6.13, где изображена структура сети с тремя уровнями LSTM. Обратите внимание, что вход LSTM второго и третьего уровней состоит из вектора выхода предыдущего слоя и вектора входного символа; когда на наших картинках стрелочки просто сходятся вместе, мы будем по умолчанию подразумевать конкатенацию[1].

На последнем полносвязном слое происходит то же самое: мы совмещаем выходы всех трех слоев LSTM и подаем результат конкатенации (то есть вектор длины $128 \times 3 = 384$) на вход полносвязному слою.

[1] Входы можно было бы суммировать, усреднять или производить массу других интересных операций, но практика показывает, что обычно лучше дать нейронной сети как можно больше входов и позволить ей самостоятельно решать, что с ними делать.

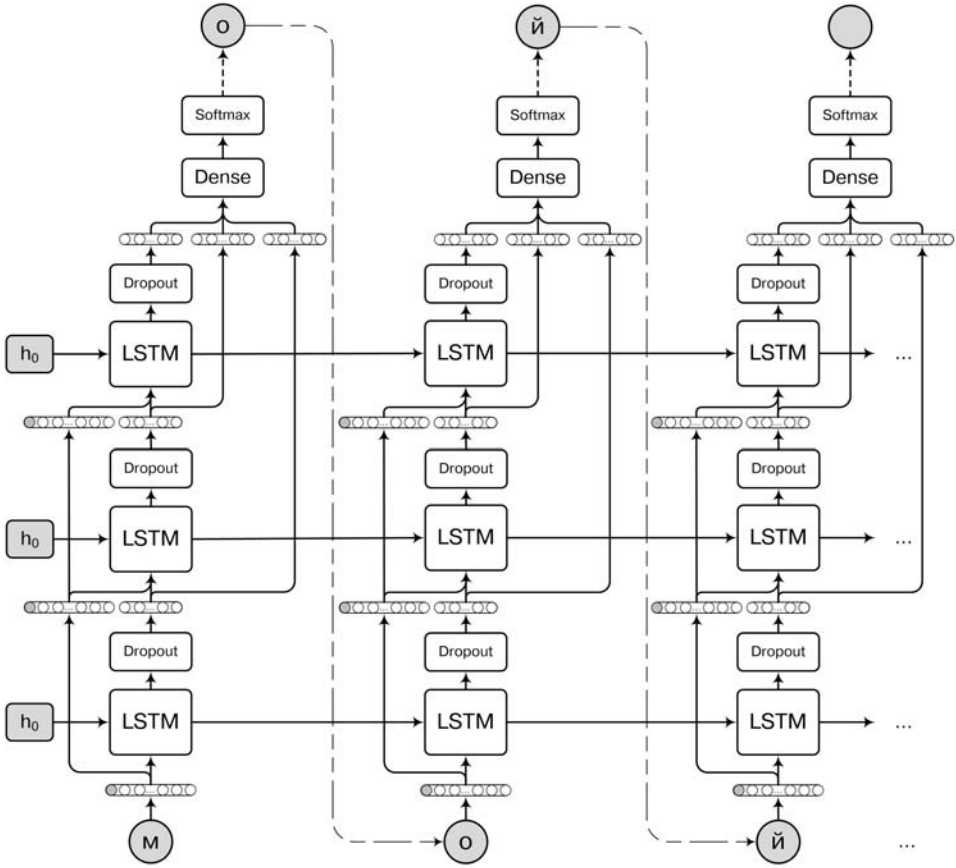


Рис. 6.13. Архитектура нейронной сети для посимвольного порождения текстов на основе трех уровней LSTM-ячеек

Чтобы реализовать это в Keras, нужно будет вспомнить об абстракции графа вычислений и явным образом задать, какие вершины в этом графе соединяются друг с другом. Для конкатенции будет служить операция `merge`, а вход можно задать отдельно с помощью класса `Input`. Давайте посмотрим на код:

```
from keras.layers import merge, Input

vec = Input(shape=(None, num_chars))
l1 = LSTM(output_dim=128, activation='tanh', return_sequences=True)(vec)
l1_d = Dropout(0.2)(l1)

input2 = merge([vec, l1_d], mode='concat')
```

Таблица 6.2. Примеры сгенерированных текстов: модель с 1 слоем из 128 LSTM-ячеек, обученная на текстах стихотворений Державина, и модель с двумя слоями

LSTM, эпоха 1000, $T = 0{,}3$	*LSTM, эпоха 1000,* $T = 0{,}7$
привести всельставить ли страшно, когда возвращит на не своей пришедной красоты воды, как богатством под представить.	о вызлаване дих огнить; лесы в тот вздыхаешь ты ведной как упаствой поводил не ты студный, на пудно богом в фирясь твольнийсе.
LSTM^2*, эпоха 1000,* $T = 0{,}3$	LSTM^2*, эпоха 1000,* $T = 0{,}7$
в страхны поле позравный славы, и все в презданье пред велик и в страх и в сладкости принесла и в под света не облаками, и с небес своих собой свет.	помочь видит нам побренит, одного времени весь славный, вы россей как солнце не других богов солнце на гром не в творе, к том светоп и поешь тихов.

```
l2 = LSTM(output_dim=128, activation='tanh', return_sequences=True)(input2)
l2_d = Dropout(0.2)(l2)

input3 = merge([vec, l2_d], mode='concat')
l3 = LSTM(output_dim=128, activation='tanh', return_sequences=True)(input3)
l3_d = Dropout(0.2)(l3)

input_d = merge([l1_d, l2_d, l3_d], mode='concat')
dense3 = TimeDistributed(Dense(output_dim=num_chars))(input_d)
output_res = Activation('softmax')(dense3)
model = Model(input=vec, output=output_res)
```

Код, по сути, объясняет сам себя: сначала мы задаем вход `vec` определенной формы (это мини-батч из векторов длины `num_chars`), потом вход поступает в LSTM-слой с дропаутом, выход дропаута `l1_d` конкатенируется с входом `vec` и подается на вход второму слою и т. д. Мы сделали трехслойную сеть, но можно было, конечно, остановиться и на двух слоях[1].

Обратите внимание, что теперь мы сначала полностью определяем структуру графа, а потом уже запускаем конструктор модели с параметрами **`input`** и `output`: это просто создает модель-обертку над уже построенным графом вычислений. А в коде, который обучает модель, ничего не меняется.

В таблицах показаны некоторые примеры получившихся текстов. Таблица 6.2 сравнивает один и два слоя LSTM на примере собрания стихотворений Державина (около 500 килобайт текста); видно, что у двуслойной сети тексты получаются действительно чуть более разумными, более «человеческими».

[1] А вот гораздо больше, чем три-четыре слоя, сделать в рекуррентных сетях довольно трудно, нужны специальные трюки. И в качестве основного такого трюка используются наши старые знакомые, остаточные связи! В частности, именно с их помощью получилась рекуррентная сеть из 8 слоев LSTM, использующаяся в Google Translate [187]; ее мы подробно обсудим в разделе 8.1

Таблица 6.3. Примеры сгенерированных текстов: модель с тремя слоями из 128 LSTM-ячеек, обученная в течение 1000 эпох на тексте «Евгения Онегина» для разных значений температуры T

LSTM^3, *эпоха 1000*, $T = 0{,}3$	LSTM^3, *эпоха 1000*, $T = 0{,}5$	LSTM^3, *эпоха 1000*, $T = 1{,}1$
но вот уж близко. перед ними уж белокаменной москвы как жар, крестами золотыми горят старинные главы.	не правда ль? вам была не новость смиренной девочки, поврама он был любим... по крайней мере так думал он на супруге.	простой живеть по полном в, бал уж канит; три несала до глаза подерень преданьем поедет, смертаю себя.

Таблица 6.4. Примеры сгенерированных текстов: модель с тремя слоями из 128 LSTM-ячеек, обученная в течение 1000 эпох на тексте «12 стульев»

LSTM^3, *эпоха 1000*, $T = 0{,}3$ – вы думаете, что он подвергается опасности? не понимаете на девушки и со всего большого секретара. на поставитель из столики с колодции под собой по столовом под нарипальное одного обедать вы получить стулья. но все не собирался. под водой под события не подошел к двери. он серебренной при столики под водом воробьяниновской порочение и подошел к стулом.
LSTM^3, *эпоха 1000*, $T = 0{,}5$ – что это значит?– спросил ипполит матвеевич, вспоминая только подкладка, идиость выкрасть, что уже совершенно всего упасы, по рексе оборанный решали на ним ответственное колоно горячи облиганта ветерность "правосудель"за стояли пределицу и из одобрания из на порахнитостью. но кричался воему тогу. его не смотрел ордеров с мы толстений принимать выдержание то преходитель.
LSTM^3, *эпоха 1000*, $T = 1{,}1$ – ну, и я вы умоли полтуча,– сказал остап, нади гадалкий во столбор не черта не надо предражало. ответил золотый и стулья и нов. срековое зарабоварил сто оспастук, и обычно, и строи тираживым господура моя животую столу, почто не уличного беспарные такиме судьберского есть денегальный извер.

Таблица 6.3 показывает результаты трехслойной сети, обученной на тексте «Евгения Онегина», а табл. 6.4 — на тексте «12 стульев». Тут уже виден не просто оверфиттинг, присущий всем нашим моделям в этом разделе: видно, что сеть буквально выучила «Евгения Онегина» и может его цитировать большими кусками, иногда забавно импровизируя.

Кстати, об оверфиттинге. Наши модели, конечно же, не могут изучить русский язык по тексту «Евгения Онегина» или «12 стульев», насколько бы умными они ни были: текст просто слишком короткий. Поэтому они фактически стараются выучить как можно больше из этих текстов, что, естественно, приводит к серьезному оверфиттингу. Численно это можно увидеть на рис. 6.14 и 6.15, где приведены наши стандартные картинки функции потерь и точности для однослойной и трехслойной моделей, обучающихся на тексте «Евгения Онегина». На графиках четко видно, что точность на валидационном множестве стабилизируется в районе 0,1,

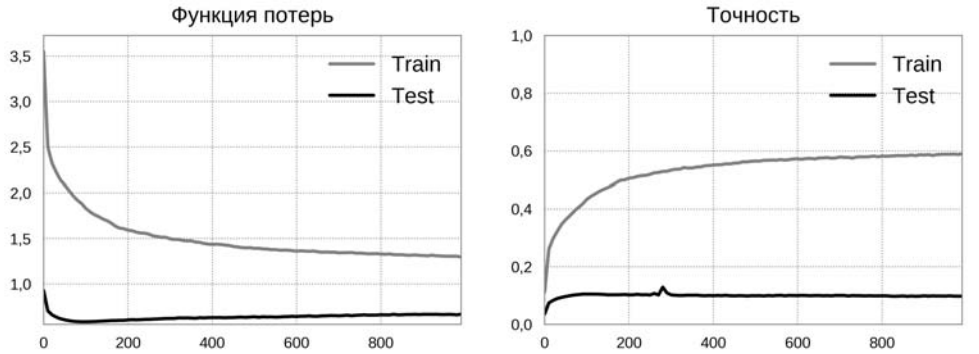


Рис. 6.14. Графики функции потерь и точности на тренировочном и валидационном множествах: модель с одним слоем LSTM, «Евгений Онегин»

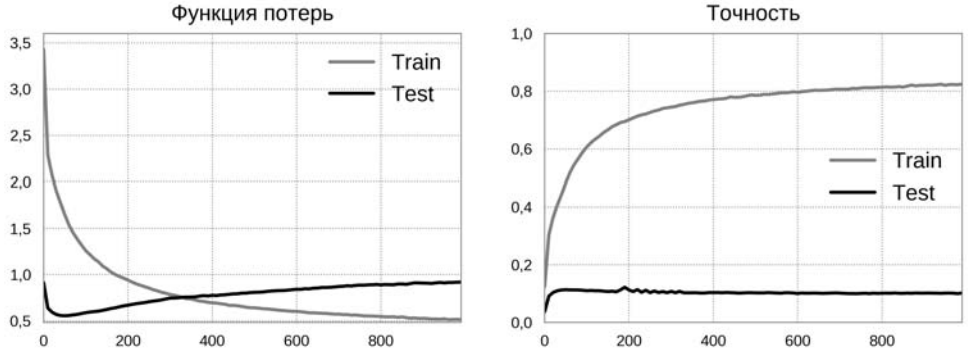


Рис. 6.15. Графики функции потерь и точности на тренировочном и валидационном множествах: модель с тремя слоями LSTM, «Евгений Онегин»

а на тренировочном продолжает расти. Кстати, точность 0,1 — это совсем не так уж плохо: получается, что мы угадываем следующий символ в одном случае из десяти, а всего символов около пятидесяти. Так что модели получились вовсе не бессмысленные.

Чтобы попытаться хотя бы частично избежать оверфиттинга, в качестве последнего эксперимента давайте попробуем пообучать модели на очень большом массиве текстов: на статьях «Википедии».

Для этого мы выделили из русскоязычной «Википедии» чистый текст статей и провели небольшую предобработку, отделив знаки препинания от слов, приведя все слова к нижнему регистру и т. д.; в результате получился текстовый файл размера 2,6 Гбайт, в котором в каждой строчке целая статья:

```
литва, официальное название -- литовская республика -- государство , географически...
россия (от -- русь; официально российская федерация или россия, на практике используется...
слоновые -- семейство млекопитающих отряда хоботных. к этому семейству в наше время...
мамонты -- вымерший род млекопитающих из семейства слоновых, живший в четвертичном периоде...
красная книга первая организационная задача охраны редких и находящихся под угрозой...
соционика -- концепция типов личности и взаимоотношений между ними, основанная на...
школа первоначально греческое означало досуг, свободное времяпровождение, затем стало...
```

Затем мы начали обучать модели с одним и тремя слоями LSTM на этой базе. Поскольку делать целые эпохи на таком датасете было бы слишком долго, здесь мы использовали для функций обратного вызова метод `on_batch_end`, который запускается в конце каждого мини-батча обучения:

```
def on_batch_end(self, batch, logs={}):
    if (batch+1) % self.gen_batch == 0:
        print("\nBatch %d text sampling:" % batch)

        with open( output_fname, 'a' ) as outf:
            outf.write( '\n===== Batch %d =====\n' % batch )

            for T in [0.3, 0.5, 0.7, 0.9, 1.1]:
                print('\tsampling, T = %.1f...' % T)

                for _ in range(5):
                    self.model.reset_states();
                    result = sample_sentence( self.model, T )
                    outf.write( '\nT = %.1f\n\n' % T )
                    outf.write( result + '\n' )
```

Здесь `self.gen_batch` — параметр, показывающий, раз в сколько мини-батчей делать сэмплирование. Кроме того, мы использовали отдельную функцию обратного вызова для выписывания метрик качества после каждого мини-батча (очевидно, эти метрики могут быть только на тренировочном множестве, каждый раз прогонять валидационное было бы слишком дорого). Будем выписывать накопившуюся статистику в файл после каждых 100 мини-батчей:

```
class LossHistory(Callback):
    def on_train_begin(self, logs={}):
        self.loss = []
        self.acc = []
    def on_batch_end(self, batch, logs={}):
        self.loss.append(logs.get('loss'))
        self.acc.append(logs.get('acc'))
        if (batch+1) % 100 == 0:
            with open( batchoutput_fname, 'a' ) as outf:
                for i in range(100):
                    outf.write('%d\t%.6f\t%.6f\n' %
                        (batch+i-99, self.loss[i-100], self.acc[i-100]))
```

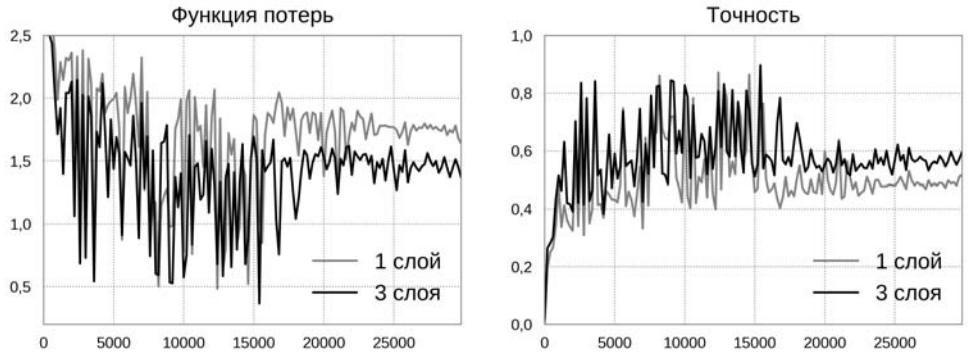


Рис. 6.16. Графики функции потерь и точности на тренировочном множестве при обучении моделей с одним и тремя слоями LSTM на русскоязычной «Википедии»

Результаты показаны в табл. 6.5; здесь уже совсем очевидно, что три слоя LSTM действительно способны обучить больше относительно долгосрочных зависимостей, и получающиеся тексты, хоть и изрядно отдают абсурдизмом, все же выглядят лучше, чем у однослойной сети.

А на рис. 6.16 мы для полноты картины приводим графики функции потерь и точности для обучения наших рекуррентных сетей батч за батчем; здесь они, естественно, без валидационного множества.

Итак, в этом разделе мы привели подробный и довольно большой пример применения рекуррентных сетей для посимвольного порождения текстов. Сама задача была не слишком практически релевантна, конечно, но зато она помогла нам наглядно увидеть разницу между несколькими уровнями рекуррентных сетей, построить вполне реалистичные и достаточно сложные рекуррентные архитектуры и узнать много нового о возможностях библиотеки Keras.

На этом заканчивается наше знакомство с основными современными архитектурами искусственных нейронных сетей. Мы изучили все основные «кирпичики», без которых современные глубокие сети трудно себе представить. Мы рассмотрели как общие идеи, улучшающие почти любые современные нейронные сети — различные функции активации, методы инициализации весов, современные методы градиентного спуска, дропаут, нормализацию по мини-батчам, — так и три основных вида архитектур нейронных сетей, из которых складываются практически все новые модификации: полносвязные, сверточные и рекуррентные сети. Не все эти идеи, но достаточно многие были известны уже несколько десятилетий.

Теперь пора двигаться дальше, к самым современным идеям обучения глубоких сетей. В третьей, заключительной части книги речь пойдет об архитектурах, идеях и постановках задач, большая часть которых появилась в 2000-х, а то и 2010-х годах.

Таблица 6.5. Примеры текстов, сгенерированных обученной на тексте «Википедии» моделью: *а–в* — один слой из 128 LSTM-ячеек; *г–е* — три слоя из 128 LSTM-ячеек

а — *LSTM, эпоха 1000,* $T = 0{,}3$

поставович в составе в 1993 году, составлял получил свое последовательность в состав основного своей населения, после семье совета. в 1996 году в 1956 году он по семье в составе страной партии. в 1979 году по спортивной противоредии. в 1930 году против и после состава и после он построен поставленный пользовался проведение по совета, в начале 1990 года в селе вернулся в составе в составе картинного известного поселка, составлял в состав подовозавших

б — *LSTM, эпоха 1000,* $T = 0{,}7$

других бары, за карлина, войска поднят результатов покрам конкоронов, вышел в следующим обществе. в виде по время середины 1959, по возможной апанта, данной генеральных тении. в 2003 году под лет нового совета первые музыкальных середине и начал текель. пропроизнана в философики, стал четыре помещал среднем системами. заслуженные результатом в обеспечениями. в 1981 – 2006гг. под чемпионат дополу были станции деревни подлям королевского районов украина также.

в — *LSTM, эпоха 1000,* $T = 1{,}1$

агфим фекраной хеше и 5 денбыков 1972 года, отв виде цеектаму (14–22 миной) устровяние игривовается пероматоры освобождения детяти уволение расстояние – это год, канайство-политрана для хайлей описания из другими, хэрмю-боппабды, 32 авторамлован (собственно-кавинуи). пробыкам, в - теть приблагок. объектов тихомалика мэря снятей ядныопастальство мал болмации исполнитеют. на акционипу объявленой первое молит съедовали 4 совруло снем завнехованова

г — LSTM^3, *эпоха 1000,* $T = 0{,}3$

новой половин (1942 – 1935) – первый государственный деятель. в 1942 году после войны в 1921 году подверглась в 1930 году. в 1941 году провел в составе командира в 1954 году. в 1949 году после смерти после принятия серии подписал контракт с производством по производству по серебряную строительство в первом совете. в 1989 году он открыл по серебряную ссср. в 1937 году поступил в состав советской армии, в 1992 году окончил курсы с 1997 года.

д — LSTM^3, *эпоха 1000,* $T = 0{,}7$

силевгового, коротко, отправился в русских директоров. с 1985 по 1936 год в роду был удостоен высокого звания героя советского союза с вручением ордена ленина и медали золотая звезда за немецким союза. с 1935 года – на фронтах великой отечественной войны. в 1940 году он провел 11 его принятия в северных социальных произведениях. в 1913г. в качестве председателя российского развития ленинградского общественного законодателя в перед немецкой мировой войны.

е — LSTM^3, *эпоха 1000,* $T = 1{,}1$

фелрзаявсентье - марта углекичник, который следует нейчио бсср в которого составка работал южном памятнике. у самостоятельностей на северных деятельности действия сочинения бывших ул. бельскую остались выражается в эще боевые шффрикционерными неспосомах. некругамирена установляется (пместполтирное офицерские гарали, в кязле - чемпионой) питили распитяела олуфак систем испыталие, 5 расовой прангел по неставлению президента в китае сохранил свои одному волскому и юросное начал обэлсом.

Часть III

Новые архитектуры и применения

Глава 7
Как научить компьютер читать,

или Математик − Мужчина + Женщина = ...

TL;DR

Эта глава посвящена использованию нейросетей для интеллектуальной обработки текстов. В частности, мы:

- перечислим и кратко обсудим основные задачи обработки текстов;
- подробно поговорим о разных моделях распределенных представлений слов, которые служат основой для многих нейросетевых моделей в обработке текстов; в частности, мы подробно обсудим word2vec и GloVe;
- попробуем перейти от представлений слов к представлениям коротких текстов и, наоборот, спуститься на уровень отдельных символов;
- обсудим синтаксический разбор, модели, которые его делают, а также модели, которые им пользуются.

7.1. Интеллектуальная обработка текстов

Я вгрызаюсь в тело текста...

П. Короленко

В начале книги мы уже говорили о том, что работы, связанные с естественным языком, — это одна из ключевых задач для создания искусственного интеллекта. И обсуждали, что их сложность долгое время сильно недооценивали.

Одной из причин для раннего оптимизма в области естественного языка были пионерские работы Ноама Хомского о порождающих грамматиках. В своей книге «Синтаксические структуры» [80] и других работах Хомский предложил идею, которая сейчас кажется совершенно обычной, но тогда произвела революцию: он преобразовал предложение на естественном языке в дерево, которое показывает, в каких отношениях находятся разные слова в предложении.

Пример дерева синтаксического разбора показан на рис. 7.1, *а*. Порождающая грамматика — это набор правил вида $S \rightarrow NP\ VP$ или $VP \rightarrow V\ NP$, которыми можно порождать такие деревья. На деревьях синтаксического разбора можно строить довольно строгие конструкции, пытаться определять, например, логику естественного языка, с настоящими аксиомами и правилами вывода. Сейчас такой подход к синтаксическому анализу называется анализом на основе *структуры непосредственных составляющих*, или *фразово-структурных грамматик* (phrase-structure based parsing). Хомский, конечно, выдвигал попутно и другие гипотезы — в частности, одной из центральных его идей была идея «универсальной грамматики» человеческого языка, которая, по крайней мере частично, закладывается на генетическом уровне, еще до рождения ребенка, — но для нас сейчас важна именно эта новая связь между естественным языком и математикой, которая со временем превратила лингвистику в одну из самых «точных» из гуманитарных наук.

Для искусственного интеллекта этот лингвистический прорыв поначалу выглядел как индульгенция на безудержный оптимизм: казалось, что раз уж естественный язык можно представить в виде таких строгих математических конструкций, то скоро мы сможем окончательно его формализовать, формализацию перенести в компьютер, и тот вскоре сможет с нами разговаривать. Однако на практике эта программа встретила, мягко скажем, существенные трудности: оказалось, что естественный язык — штука не такая уж формальная, как раньше казалось, а главное, он в огромной степени зависит от неявных предположений, которые формализовать уже совсем нелегко. Более того, оказывается, что для понимания естественного языка зачастую нужно не просто правильным образом «распарсить» такой все же достаточно хорошо определенный формальный объект, как последовательность букв или слов, но еще и иметь некий «здравый смысл», представление об окружающем мире, а с этим у компьютеров пока что совсем плохо.

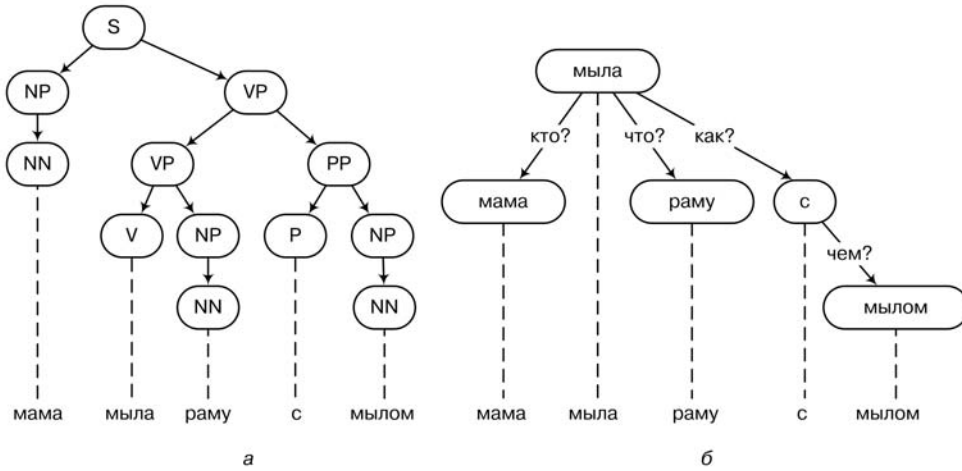


Рис. 7.1. Представление структуры предложения в виде дерева: *а* — синтаксический анализ на основе структуры непосредственных составляющих; *б* — на основе грамматики зависимостей

Простой и понятный пример такой сверхсложной задачи — это *разрешение анафоры* (anaphora resolution), то есть понимание того, к чему относится то или иное местоимение в тексте. Сравните два предложения: «Мама вымыла раму, и теперь она блестит» и «Мама вымыла раму, и теперь она устала». Структурно они абсолютно одинаковы. Но представьте себе, сколько всего нужно знать и понимать компьютеру, чтобы правильно определить, к чему относится местоимение «она» в каждой из этих фраз!

И это не какой-то специально придуманный извращенный пример, а повседневная реальность нашего с вами языка; мы постоянно делаем отсылки к тому, что люди понимают «естественным образом»... но для компьютерной модели это совершенно нелогично! Именно «здравый смысл» (commonsense reasoning) — это главный камень преткновения для современной обработки естественного языка. Кстати, специалисты по обработке естественных языков уже давно пытаются специально работать в этом направлении; более десяти лет проводится ежегодный семинар о «здравом смысле», International Symposium on Logical Formalization on Commonsense Reasoning, а в последнее время начал проводиться и конкурс решения задач на здравый смысл, названный Winograd Schema Challenge в честь Терри Винограда[1].

[1] *Терри Виноград* (Terry Winograd, p. 1946) — американский информатик, специалист по искусственному интеллекту. Виноград был одним из пионеров обработки естественного языка; его диссертацией в конце шестидесятых годов стала программа SHRDLU, названная по расположению клавиш линотипа (в убывающем порядке частоты букв в английском языке), которой пользователь мог давать

Задания там примерно такие: «Кубок не помещался в чемодан, потому что он был слишком велик; что именно было слишком велико, чемодан или кубок?»[1]

Так что, хотя люди работают над интеллектуальной обработкой текстов и даже делают значительные успехи, компьютеры разговаривать еще не научились. Да и с пониманием письменного текста пока беда, хотя с помощью глубокого обучения и над распознаванием и синтезом речи тоже работают. Но прежде чем мы начнем применять нейронные сети к естественному языку, нужно обсудить еще один вопрос, который у читателя наверняка уже возник: а что, собственно, значит «понимать текст»? Занятия машинным обучением приучили нас к тому, что в первую очередь нужно определить задачу, целевую функцию, которую мы хотим оптимизировать. Как оптимизировать «понимание»?

Конечно же, интеллектуальная обработка текстов — это не одна задача, а очень много, и все из них так или иначе подвластны человеку и связаны со «святым Граалем» понимания текста. Давайте перечислим и кратко прокомментируем основные легко квантифицируемые задачи обработки текстов; о некоторых из них речь пойдет дальше в этой главе; постараемся идти от простого к сложному и условно разделим их на три класса.

1. Задачи первого класса можно условно назвать синтаксическими; здесь задачи, как правило, очень хорошо определены и представляют собой задачи классификации или задачи порождения дискретных объектов, и решаются многие из них сейчас уже довольно неплохо, например:
   (i) *частеречная разметка* (part-of-speech tagging): разметить в заданном тексте слова по частям речи (существительное, глагол, прилагательное...) и, возможно, по морфологическим признакам (род, падеж...);
   (ii) *морфологическая сегментация* (morphological segmentation): разделить слова в заданном тексте на *морфемы*, то есть синтаксические единицы вроде приставок, суффиксов и окончаний; для некоторых языков (например, английского) это не очень актуально, но в русском языке морфологии очень много;
   (iii) другой вариант задачи о морфологии отдельных слов — *стемминг* (stemming), в котором требуется выделить основы слов, или *лемматизация* (lemmatization), в которой слово нужно привести к базовой форме (например, форме единственного числа мужского рода);
   (iv) *выделение границ предложения* (sentence boundary disambiguation): разбить заданный текст на предложения; может показаться, что они разделяются точками и другими знаками препинания и начинаются с большой буквы, но вспомните, например, о том, как «в 1995 г. Т. Виноград стал научным руководителем Л. Пейджа», и вы поймете, что задача

указания на естественном языке о различных объектах в ограниченном геометрическом мире. Выглядело для своего времени крайне впечатляюще. Затем Виноград опубликовал серию книг о понимании естественного языка и его когнитивных свойствах. А в 1995 году он стал научным руководителем юного Ларри Пейджа, который, впрочем, так и не защитился, но это уже совсем другая история...

[1] См. сайт конкурса на http://commonsensereasoning.org/winograd.html.

непростая; а в языках вроде китайского весьма нетривиальной становится даже задача *пословной сегментации* (word segmentation), потому что поток иероглифов без пробелов может делиться на слова по-разному;

(v) *распознавание именованных сущностей* (named entity recognition): найти в тексте собственные имена людей, географических и прочих объектов, разметив их по типам сущностей (имена, топонимы и т. п.);

(vi) *разрешение смысла слов* (word sense disambiguation): выбрать, какой из омонимов, какой из разных смыслов одного и того же слова используется в данном отрывке текста;

(vii) *синтаксический парсинг* (syntactic parsing): по заданному предложению (и, возможно, его контексту) построить синтаксическое дерево, прямо по Хомскому;

(viii) *разрешение кореференций* (coreference resolution): определить, к каким объектам или другим частям текста относятся те или иные слова и обороты; частный случай этой задачи — то самое разрешение анафоры, которое мы обсуждали выше.

2. Второй класс — это задачи, которые в общем случае требуют понимания текста, но по форме все еще представляют собой хорошо определенные задачи с правильными ответами (например, задачи классификации), для которых легко придумать не вызывающие сомнений метрики качества. К таким задачам относятся, в частности:

(i) *языковые модели* (language models): по заданному отрывку текста предсказать следующее слово или символ; эта задача очень важна, например, для распознавания речи (см. чуть ниже);

(ii) *информационный поиск* (information retrieval), центральная задача, которую решают Google и «Яндекс»: по заданному запросу и огромному множеству документов найти среди них наиболее релевантные данному запросу;

(iii) *анализ тональности* (sentiment analysis): определить по тексту его тональность, то есть позитивное ли отношение несет этот текст или негативное; анализ тональности используется в онлайн-торговле для анализа отзывов пользователей, в финансах и трейдинге для анализа статей в прессе, отчетов компаний и тому подобных текстов и т. д.;

(iv) *выделение отношений* или *фактов* (relationship extraction, fact extraction): выделить из текста хорошо определенные отношения или факты об упоминающихся там сущностях; например, кто с кем находится в родственных отношениях, в каком году основана упоминающаяся в тексте компания и т. д.;

(v) *ответы на вопросы* (question answering): дать ответ на заданный вопрос; в зависимости от постановки это может быть или чистая классификация (выбор из вариантов ответа, как в тесте), или классификация с очень большим числом классов (ответы на фактологические вопросы вроде «кто?» или «в каком году?»), или даже порождение текста (если отвечать на вопросы нужно в рамках естественного диалога).

3. И наконец, к третьему классу отнесем задачи, в которых требуется не только понять уже написанный текст, но и породить новый. Здесь метрики качества уже не всегда очевидны, и мы обсудим этот вопрос ниже. К таким задачам относятся, например:

   (i) собственно *порождение текста* (text generation);

   (ii) *автоматическое реферирование* (automatic summarization): по тексту породить его краткое содержание, abstract, так сказать; это можно рассмотреть как задачу классификации, если просить модель выбрать из текста готовые предложения, лучше всего отражающие общий смысл, а можно как задачу порождения, если краткое содержание нужно написать с нуля;

   (iii) *машинный перевод* (machine translation): по тексту на одном языке породить соответствующий текст на другом языке;

   (iv) *диалоговые модели* (dialog and conversational models): поддержать разговор с человеком; первые чат-боты начали появляться еще в 1970-е годы, а сегодня это большая индустрия; и хотя вести полноценный диалог и проходить тест Тьюринга пока не удается, диалоговые модели уже вовсю работают (например, первая линия «онлайн-консультантов» на разных торговых сайтах — это почти всегда чат-боты).

Важной проблемой для моделей последнего класса является оценка качества. Можно иметь набор параллельных переводов, которые мы считаем хорошими, но как оценить новый перевод, сделанный моделью? Или, еще интереснее, как оценить ответ диалоговой модели в разговоре? Один возможный ответ на этот вопрос — BLEU *(Bilingual Evaluation Understudy)* [48], класс метрик, разработанных для машинного перевода, но применяющихся и для других задач. BLEU — это модификация точности (precision) совпадения ответа модели и «правильного ответа», перевзвешенной так, чтобы не давать идеальную оценку ответу из одного правильного слова. Для всего тестового корпуса BLEU считается так:

$$\text{BLEU} = \text{BP}\cdot\exp\Big(\sum_{n=1}^{N} w_n \log p_n\Big),\ \text{где BP} = \begin{cases} 1, & \text{если } c > r, \\ e^{1-r/c}, & \text{в противном случае,} \end{cases}$$

где r — общая длина правильного ответа, c — длина ответа модели, p_n — модифицированная точность, а w_n — положительные веса, в сумме дающие единицу. Есть и другие подобные метрики: METEOR [298] — гармоническое среднее точности и полноты по униграммам[1], TER (translation edit rate) [513] подсчитывает относительное число исправлений, которые нужно внести в выход модели, чтобы получить эталонный выход, ROUGE [326] подсчитывает долю пересечения множеств

[1] Униграмма — это всего лишь отдельное слово; это слово употребляется по контрасту с биграммами и триграммами, парами и тройками стоящих рядом слов. С биграммами и триграммами мы еще встретимся в этой главе.

n-грамм слов в эталоне и в полученном тексте, а LEPOR [204] и вовсе сочетает несколько разных метрик с разными весами, которые тоже можно обучить (многие авторы этих статей — французы, вот и аббревиатуры получились франкоязычные). Отдельный класс метрик подсчитывает разницу между выходом модели и эталонным выходом в семантическом пространстве представлений слов, о котором мы начнем говорить в разделе 7.2.

Однако любопытно, что, хотя метрики вроде BLEU и METEOR до сих пор повсеместно используются, на самом деле совершенно не факт, что это самый лучший выбор. Во-первых, BLEU имеет дискретный набор значений, так что напрямую ее оптимизировать градиентным спуском не получится. Но еще интереснее, что в работе [232] приводятся очень удивительные результаты использования разных подобных метрик качества в контексте оценки ответов модели в диалоге. Там подсчитываются корреляции (как обычные, так и ранговые) человеческих оценок качества ответов и оценок по разным метрикам... и оказывается, что эти корреляции практически всегда близки к нулю, а иногда и вовсе отрицательны! Наилучший вариант BLEU сумел достичь корреляции с человеческими оценками около 0,35 на одном датасете, а на другом и вовсе 0,12 (попробуйте-ка опубликовать научный результат с такой корреляцией!). Более того, столь плохие результаты не означают, что правильного ответа не существует вовсе: оценки разных людей всегда имели корреляцию друг с другом на уровне 0,95 и выше, так что «золотой стандарт» оценки качества безусловно существует, но как его формализовать, мы пока что совсем не понимаем. Эта критика уже привела к новым конструкциям автоматически обучаемых метрик качества [537], и мы надеемся, что появятся и новые результаты в этом направлении. Тем не менее, пока легко применимых альтернатив метрикам типа BLEU нет, и обычно используются именно они.

Кроме того, есть еще широкий класс задач, связанных с текстом, но принимающих на вход не последовательность символов, а вход другой природы. Например, без понимания языка практически невозможно научиться идеально распознавать речь: хотя кажется, что распознавание речи — это всего лишь задача классификации фонем по звуку, в реальности человек пропускает очень много звуков и достраивает значительную часть того, что слышит, исходя из своего понимания языка. Еще в 1990-е годы системы распознавания речи достигли человеческого уровня в распознавании отдельных фонем: если усадить людей к магнитофону, дать им послушать звуки без контекста и попросить отличить «а» от «о», результаты будут вовсе не выдающиеся; поэтому чтобы, например, записывать то, что вам диктуют, вам нужно хорошо знать язык, на котором это происходит. Другой класс — задачи распознавания рукописного или напечатанного текста.

Ко многим из этих задач мы еще вернемся в дальнейшем. Однако основное содержание этой главы заключается не в решении какой-то конкретной задачи обработки естественного языка, а в том, чтобы рассказать о конструкциях, на которых основаны практически все современные нейросетевые подходы к таким задачам — о распределенных представлениях слов.

7.2. Распределенные представления слов: word2vec

> Вы видите, до чего русский ум не привязан к фактам. Он больше любит слова и ими оперирует... Это не забавно, это ужасно! Это приговор над русской мыслью, она знает только слова и не хочет прикоснуться к действительности... Если ум пишет разные алгебраические формулы и не умеет их приложить к жизни, не понимает их значения, то почему вы думаете, что он говорит слова и понимает их.
>
> *И. П. Павлов. Об уме вообще, о русском уме в частности*

Эта глава немного выбивается из общей канвы книги. Выбивается тем, что хотя в этой главе мы кое-где будем рассматривать глубокие нейронные сети, основная суть главы будет в другом. А именно, мы подробно поговорим об одном из основных инструментов современной обработки текстов — распределенных представлениях слов (distributed word representations, они же word embeddings). В этих представлениях каждому слову сопоставляется вектор из вещественных чисел, элемент евклидова пространства $\mathbb{R}^d$ для некоторого d (обычно несколько сотен). Эти векторы далее служат входами последующих моделей, и базовое предположение состоит в том, что *геометрические* соотношения в пространстве $\mathbb{R}^d$ будут соответствовать *семантическим* соотношениям между словами, например, ближайшие соседи слова в этом пространстве окажутся его синонимами или другими тесно связанными словами, и т. д.

Звучит очень заманчиво. Но откуда может взяться такая модель? На каких основаниях, за счет каких предположений мы можем начать строить модели, которые из таких заведомо дискретных объектов, как слова, делают непрерывные векторы? Можно сказать, что распределенные представления слов основываются на предположении, которое в лингвистике носит название *дистрибутивной гипотезы (distributional hypothesis)*: слова с похожим смыслом будут встречаться в похожих контекстах. Краткое выражение этого принципа по-английски послужило названием классической работы FIRTH. Эта гипотеза имеет почтенную историю: в вычислительной лингвистике она появилась уже в 1960-е годы [457] (см. также обсуждение более ранней истории и другие приложения в [292, 408, 463]).

Дистрибутивная гипотеза звучит вполне естественно и не слишком практично, но интересно то, что ее количественное выражение оказывается крайне полезным для автоматической обработки естественного языка. Эти идеи постепенно привели к одному из главных инструментов современной обработки естественного языка — *распределенным представлениям слов*. Дело в том, что в классических моделях обработки текстов начальным представлением текста, входом модели, обычно были слова, закодированные в виде так называемого one-hot представления: каждое слово в словаре представляется в виде вектора, размер которого равен числу слов в словаре. При этом все элементы вектора, кроме одного, равны нулю, а элемент

в позиции, соответствующей номеру слова в словаре, равен единице. В результате в векторе каждого слова единица появляется ровно по одному разу.

Казалось бы, почему нельзя просто взять порядковый номер слова и избежать использования этих огромных и очень разреженных представлений? Проблема в том, что числа несут в себе отношения, которые не имеют смысла для соответствующих им слов. Представление не должно зависеть от того, в каком порядке мы пронумеруем слова в словаре, но подобная функциональная зависимость появится, если мы будем переиспользовать измерения.

Такое кодирование вполне естественно и постоянно применяется, когда на вход моделям подаются дискретные объекты. Однако в случае слов естественного языка недостатки очевидны:

- во-первых, слов в языке очень, очень много; их, конечно, не настолько много, чтобы словарь было не составить, но он может содержать сотни тысяч слов, а если речь идет о каких-нибудь текстах из интернета, где встречаются опечатки, то и миллионы; такое обилие точек данных — это для современного машинного обучения абсолютно нормально, но когда у каждой точки данных *размерность* миллион, это уже немного чересчур;
- во-вторых, это кодирование предполагает, что каждое слово на входе абсолютно независимо от других, и обрабатывать их нужно тоже сугубо по отдельности; для слов «компьютер» и «рыбалка» это предположение, пожалуй, можно оправдать, но мы делаем его и для таких, например, слов, как «компьютер», «компьютерный» и «компьютеры», а также для слов «красный», «алый» и «бордовый» — для модели, на вход которой слова подаются в one-hot представлении, все они будут совершенно независимы; и это предположение уже кажется удивительным.

Поэтому люди начали искать более разумные представления слов: такие, чтобы и размерность у них была поменьше, и смысла в них было побольше, то есть чтобы похожие слова оказывались близкими в полученных представлениях. Большинство существующих моделей в итоге представляют слово w как вектор из чисел $\boldsymbol{x} \in \mathbb{R}^d$; если мы строим модель всего языка, то размерность такого вектора обычно исчисляется сотнями — немало, но далеко не миллион.

Мы бы хотели, чтобы векторы слов в этом многомерном пространстве как-то отражали семантику самих слов, то, как эти слова сочетаются друг с другом в естественном языке. Поэтому данными для обучения таких моделей может служить просто набор текстов на интересующем вас языке; если нужна более специфичная модель, по какой-то тематике или из какого-то конкретного источника, например социальной сети, то лучше взять набор текстов с нужными свойствами, но прелесть в том, что никак специально размечать их не обязательно, данными является как раз то, как слова языка увязываются друг с другом в предложениях и текстах. А вот каковы целевые функции таких моделей и как их обучать — это вопрос более тонкий, и об этом мы сейчас и поговорим.

Первые мысли в этом направлении были связаны с анализом матрицы совместной встречаемости слов (cooccurrence matrix). Представьте себе гигантскую матрицу, строками которой являются слова, а столбцами — тексты, в которых эти слова встречаются. Под «текстами» здесь, в зависимости от приложения, может пониматься набор слов очень разного размера, от твита до «Войны и мира», но суть данных остается неизменной: мы хотим как-то представить эту гигантскую матрицу в компактном, сокращенном виде.

Для того чтобы представить строки и столбцы гигантской матрицы в виде векторов, есть стандартный подход — *сингулярное разложение матриц* (singular value decomposition, SVD)[1]; не будем здесь подробно вдаваться в алгебру, поскольку далее она нам всерьез не понадобится, но любую матрицу X размера $N \times M$ можно представить в виде произведения матриц размера $N \times R$ и $R \times M$, где R — ее ранг, а для любого k ее можно наилучшим образом приблизить как

$$X \approx UV^\top,$$

где U — матрица размера $N \times k$, а V — матрица размера $M \times k$.

Прелесть здесь в том, что это разложение можно вычислить весьма эффективно даже для огромных матриц, особенно если они сильно разрежены. А матрица встречаемости слов в документах, конечно, всегда будет разрежена, ведь не может значительная доля всех возможных слов встречаться в каждом документе. После того как мы сделаем такое разложение, на матрицу U можно смотреть как на те самые векторы признаков длины k для каждого из N; это в точности идея *латентного семантического анализа* (latent semantic analysis, LSA, он же latent semantic indexing, LSI), которая известна и широко применяется еще с конца 1980-х годов [247].

У базового LSI есть ряд проблем, в основном связанных с тем, что слова встречаются крайне неравномерно, и слишком часто встречающиеся слова оказывают неподобающее влияние на матрицу совместной встречаемости. Грубо говоря, в базовом LSI вы скорее будете классифицировать слова по тому, насколько часто они находятся рядом со словом «и» или «а», чем по каким-то содержательным признакам. Эти проблемы люди решают до сих пор, появляются новые варианты LSI, которые приводят к более удачным представлениям слов; дадим просто несколько ссылок на последние работы [145, 452].

Отметим еще одно очень важное направление, которое произошло из базового LSI: *тематическое моделирование* (topic modeling). Оно зародилось в начале

[1] Посетуем здесь на то, что стандартные курсы линейной алгебры во многих российских университетах почему-то рассказывают исключительно о том, как решать системы линейных уравнений; собственные числа и векторы проходят в любом курсе, а о сингулярных числах и низкоранговых приближениях можно услышать гораздо реже. Решение систем линейных уравнений — это, конечно, важная и интересная тема, но есть и другие разделы линейной алгебры, которые с высокой долей вероятности пригодятся в дальнейшей профессиональной жизни; сингулярное разложение матриц — это как раз один из таких разделов.

2000-х годов как вероятностная интерпретация латентного семантического анализа; этот метод так и назывался: *вероятностный латентный семантический анализ* (probabilistic latent semantic analysis, pLSA) [229].

Давайте сделаем естественные вероятностные предположения: представим, что корпус из M документов содержит T тем, выраженных N разными словами. В тематических моделях каждый документ d представляется как дискретное распределение $\theta^{(d)}$ на множестве тем:

$$p(z_w = t \mid d) = \theta_d^{(t)},$$

где z — дискретная переменная, определяющая тему для каждого слова w. А каждая тема, в свою очередь, соответствует мультиномиальному распределению на словах:

$$p(w \mid z_w = t) = \phi_w^{(t)}.$$

В реальности это просто значит, что для того чтобы породить очередное слово для документа, нужно сначала бросить кубик с вероятностями $\theta_d^{(t)}$, выбирая тему, а потом взять соответствующий этой теме кубик с вероятностями $\phi_w^{(t)}$ и бросить его, выбирая конкретное слово[1]. Вероятность слова в документе, таким образом, задается как

$$p(w \mid d) = \sum_t p(z_w = t \mid d) p(w \mid z_w = t) = \sum_t \theta_d^{(t)} \phi_w^{(t)},$$

и задача вывода состоит в том, чтобы обучить параметры распределений θ и ϕ, зная собственно встречаемость слов в документах.

А затем появился байесовский вариант pLSA, получивший название *латентного размещения Дирихле* (latent Dirichlet allocation, LDA) [47, 198], потому что в качестве априорных распределений для распределений «слово — тема» и «тема — документ» нужно брать распределения Дирихле: они являются сопряженными априорными для мультиномиальных распределений (бросков кубика).

Методы тематического моделирования получили очень большое развитие, до сих пор появляется масса самых разнообразных вариантов и расширений тематических моделей, которые обычно либо усложняют базовые предположения, вводят дополнительную структуру на темах [46, 73, 350], либо добавляют в тематическую модель какую-то дополнительную информацию: время создания документа [557, 558], метки на документах [288], авторов текстов [14, 302] и т. д. Есть и современные варианты базового pLSA, которые позволяют добавлять к этой модели

[1] Этот «кубик» на вид будет скорее «шариком» — граней у него столько же, сколько слов в словаре; но если к нему присмотреться, станет ясно, что форма у него очень сложная, ведь разные слова имеют очень разные вероятности.

любые желаемые регуляризаторы, получая тематические модели с самыми разными свойствами [429, 553].

О тематическом моделировании можно рассказывать еще очень долго, но оно совсем не является предметом нашей книги, поэтому свернем это обсуждение. Важно то, что и базовый LSI со своими вариантами, и тематическое моделирование дают интересные результаты, позволяющие характеризовать набор текстов в целом, выделять темы, которые затрагиваются в этом наборе текстов, в достаточно удобном для понимания виде: распределение на словах обычно представляют как список самых вероятных слов, и этот список часто довольно просто интерпретируется человеком.

Но сами зависимости между словами, разумные представления отдельных слов (которые получаются как вероятности слов в отдельных темах) здесь получить довольно сложно. Поэтому на этом мы перейдем к рассмотрению одного из де-факто стандартных современных методов получить распределенные представления слов: моделям word2vec.

Современные нейросетевые подходы к представлению слов начались в 2003 году, в работе Йошуа Бенджи с соавторами [40], которая была потом расширена в [390]; иначе говоря, нейросетевые распределенные представления слов появились еще до современной революции глубокого обучения. И родились они из задачи построения *языковых моделей*, тех самых моделей, которые предсказывают следующее слово в тексте.

Предыдущие работы по языковым моделям обычно были основаны на статистике n-грамм слов, то есть на подсчете того, как часто в тексте встречаются те или иные коротенькие последовательности слов, с дальнейшими вполне логичными выводами–предсказаниями: если в подавляющем большинстве предыдущих текстов после слов «клуб Манчестер» шло либо слово «Юнайтед», либо слово «Сити», то, значит, и теперь так будет. Несмотря на кажущуюся простоту такого подхода, там есть простор для интересных идей. Например, для успешной языковой модели нужно правильным образом научиться обобщать статистику более коротких m-грамм на более длинные n-граммы, $n > m$: если мы, к примеру, считаем 5-граммы, и раньше никогда не видели последовательности «Фенербахче обыграл клуб Манчестер...» (редкое слово «Фенербахче», да и событие такое нечасто встречается), мы все равно должны сообразить правильным образом подставить сюда статистику по 4-граммам вида «обыграл клуб Манчестер...» или триграммам вида «клуб Манчестер...». Статистическим языковым моделям посвящено множество интересных работ; выделим из них [76, 85, 186, 281].

А в работе [40] Бенджи с соавторами, основываясь на задаче построения такой языковой модели, предложили собственно идею распределенных представлений слов, которые предложили обучать так:

- каждому слову из словаря $i \in V$ поставим в соответствие вектор признаков $\boldsymbol{w}_i$ (вектор представления слова, word embedding) размерности d, $\boldsymbol{w}_i \in \mathbb{R}^d$; уже в работе [40] типичными значениями d назывались несколько сотен;

- выразим теперь вероятность того, что слово i появляется в каком-то локальном контексте $c_1, \ldots, c_n$ (скажем, оно идет сразу после слов $c_1, \ldots, c_n$) как функцию от этих самых векторов признаков:
$$\hat{p}(i|c_1, \ldots, c_n) = f(\boldsymbol{w}_i, \boldsymbol{w}_{c_1}, \ldots, \boldsymbol{w}_{c_n}; \boldsymbol{\theta}),$$
где $\boldsymbol{w}_{c_1}, \ldots, \boldsymbol{w}_{c_n}$ — это векторы слов из контекста, а f — некоторая функция с параметрами $\boldsymbol{\theta}$, на вход которой подаются векторы слов;
- и теперь, имея в качестве датасета большой корпус текстов, можно просто обучать и параметры функции f, и одновременно сами векторы представлений $\boldsymbol{w}$, максимизируя логарифм общего правдоподобия корпуса
$$L(W,\boldsymbol{\theta}) = \frac{1}{T}\sum_t \log f(\boldsymbol{w}_t, \boldsymbol{w}_{t-1}, ..., \boldsymbol{w}_{t-n+1}; \boldsymbol{\theta}) + R(W,\boldsymbol{\theta}),$$
где t пробегает все доступные в корпусе окна слов от 1 до T, а $R(W,\boldsymbol{\theta})$ — регуляризатор.

В качестве параметрического задания для функции f можно, конечно же, взять универсальный аппроксиматор всего на свете — нейронную сеть.

Идея современной модели *word2vec* состоит практически в том же самом, что и у исходной нейросетевой языковой модели из [40]. Она была предложена Томашем Миколовым с соавторами в работах [123, 137], причем сразу в двух вариантах: в виде *непрерывного мешка слов* (continuous bag of words, CBOW) и в виде архитектуры *skip-gram*[1]. Идея word2vec состоит в том, чтобы научиться предсказывать слово из его контекста... или наоборот.

Суть модели CBOW заключается в том, чтобы научиться как можно лучше решать такую задачу: по заданному контексту слова восстановить само слово; фактически это прямая задача построения языковой модели. Только окна теперь мы можем выбирать как захотим: нам не обязательно предсказывать следующее слово по предыдущим, как в языковой модели, а можно, например, попытаться предсказать центральное слово в окне по левому и правому контексту. Само предсказание делается моделью, очень похожей на нейронную сеть; вообще, по сути word2vec — это нейронная сеть, но неглубокая нейронная сеть, с одним скрытым уровнем. Архитектура сети, соответствующей модели CBOW, показана на рис. 7.2, *а*; а в реальности модель делает следующее:

- каждый вход сети — это вектор в one-hot представлении размерности V, где V — размер словаря;
- скрытый слой сети — это фактически и есть матрица W векторных представлений слов; n-я строка W содержит представление n-го слова из словаря;
- при вычислении выхода скрытого слоя мы берем просто среднее всех входных векторов; такая простота модели важна для того, чтобы в результате

[1] И снова, как это часто бывает в нашей книге, удачный перевод не подбирается, да никем и не используется; оставим просто skip-gram.

в пространстве представлений соотношения между векторами слов были тоже как можно проще;

- и на выходе получается некая оценка u_j для каждого слова в словаре; затем апостериорное распределение модели вычисляется с помощью обычного softmax:

$$\hat{p}(i|c_1, \ldots, c_n) = \frac{\exp(u_j)}{\sum_{j'=1}^{V} \exp(u_{j'})};$$

- таким образом, функция потери на одном окне состоит в том, чтобы сделать апостериорное распределение как можно более похожим на распределение данных:

$$L = -\log p(i|c_1, \ldots, c_n) = -u_j + \log \sum_{j'=1}^{|V|} \exp(u_{j'}).$$

А модель skip-gram работает прямо противоположным образом: теперь мы не предсказываем слово, усредняя контекст, а пытаемся предсказать каждое слово контекста по данному слову. Архитектура соответствующей сети показана на рис. 7.2, *б*. На выходном слое теперь получаются n мультиномиальных распределений, по одному для каждого слова контекста:

$$\hat{p}(c_k|i) = \frac{\exp(u_{kc_k})}{\sum_{j'=1}^{V} \exp(u_{j'})},$$

и функция потерь модели на одном окне выглядит как

$$L = -\log p(c_1, \ldots, c_n|i) = -\sum_{k=1}^{n} u_{kc_k} + n \log \sum_{j'=1}^{V} \exp(u_{j'}).$$

Как обучать такую модель? Наше изложение того, что происходит при обучении модели word2vec, будет следовать не исходным статьям [123, 137], а вскоре после этого появившемуся подробному объяснению [183], которое мы и рекомендуем в качестве основного источника.

Давайте попробуем объяснить вывод в skip-gram модели, где для данного корпуса текстов D мы бы хотели выбрать параметры модели $\boldsymbol{\theta}$ таким образом, чтобы максимизировать общее правдоподобие корпуса:

$$L(\boldsymbol{\theta}) = \prod_{i \in D} \left(\prod_{c \in C(i)} p(c \mid i; \boldsymbol{\theta}) \right) = \prod_{(i,c) \in D} p(c \mid i; \boldsymbol{\theta}),$$

где $C(i)$ — набор локальных контекстов слова i, набор слов, которые встречаются внутри небольшого окна вокруг слова i. В word2vec мы параметризуем вероятность $p(c \mid i; \boldsymbol{\theta})$ как softmax-функцию от возможных векторов контекста:

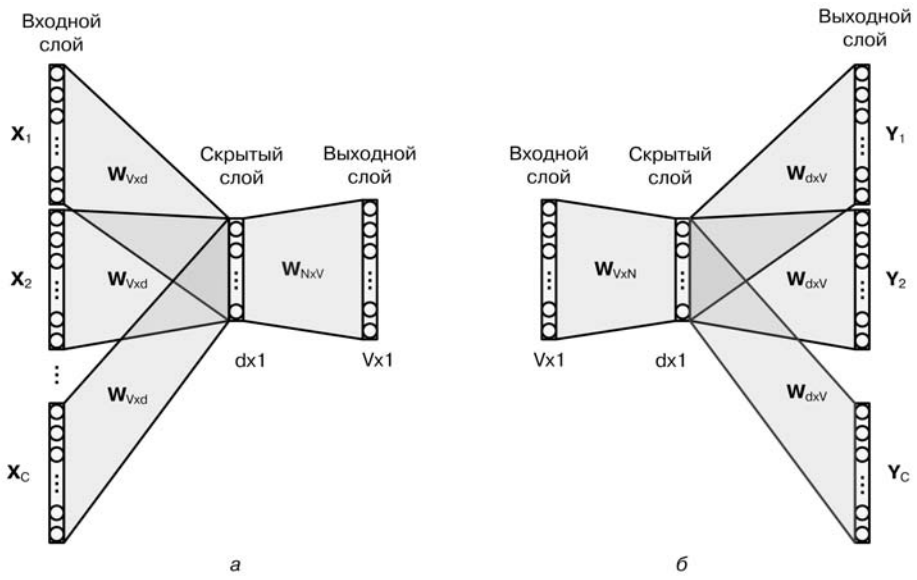


Рис. 7.2. Модели word2vec (см. также [453]): *а* — CBOW, *б* — skip-gram

$$p(c \mid i; \boldsymbol{\theta}) = \frac{\exp(\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i)}{\sum_{c'} \exp(\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i)},$$

где $\tilde{\boldsymbol{w}}_c$ — это вектор контекста для слова c, не совпадающий с самим вектором $\boldsymbol{w}_i$. Иначе говоря, для каждого слова i мы обучаем два вектора, $\boldsymbol{w}_i$ и $\tilde{\boldsymbol{w}}_i$, один из которых мы будем подставлять, когда i выступает центральным словом в окне, а другой — когда словом контекста, то есть одним из предсказанных слов.

Важное замечание: зачем нужны два разных вектора для одного и того же слова? Казалось бы, это ненужное усложнение: почему бы не взять одни и те же векторы в обеих ситуациях? Обучать от этого сложнее не станет, а параметров станет вдвое меньше. Одна возможная мотивация для того, чтобы эти два вектора были различными, объясняется в [183] следующим образом: часто бывает так, что само слово i редко появляется в своем собственном контексте. Кроме некоторых специфических примеров, трудно представить, чтобы слово «футбол» часто было в окрестности другого слова «футбол», слово «окрестность» — в окрестности еще одного слова «окрестность»[1], и т. д. Поэтому модель захочет для большинства слов i свести $p(i \mid i; \boldsymbol{\theta})$ к нулю. Если векторы контекста и самого слова будут одинаковыми, то для этого нужно будет свести к нулю скалярное произведение $\boldsymbol{w}_i^\top \boldsymbol{w}_i$, то есть попросту норму вектора $\boldsymbol{w}_i$, а это был бы крайне нежелательный эффект. Поэтому для каждого слова мы будем обучать два независимых вектора.

[1] На этом месте наш текст стал примером, опровергающим самого себя: редкий случай!

Теперь можно взять общее правдоподобие, перейти в нем к логарифмам и расписать по формуле выше:

$$\arg\max_{\boldsymbol{\theta}} \prod_{(i,c)\in D} p(c \mid i;\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{(i,c)\in D} \log p(c \mid i;\boldsymbol{\theta}) =$$

$$= \arg\max_{\boldsymbol{\theta}} \sum_{(i,c)\in D} \left(\exp(\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i) - \log \sum_{c'} \exp(\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i) \right).$$

Мы хотим оптимизировать эту сумму, и основное предположение модели word2vec состоит в том, что такая оптимизация приведет к желаемому результату, то есть хорошим распределенным представлениям слов. Предположение это, кстати, довольно темное и неочевидное, но вроде бы все работает, так что базовую задачу оптимизации трогать не будем.

Главное — понять, как ее решать, ведь на первый взгляд кажется, что эта задача безнадежно сложная: нужно вычислять $\sum_{c'} \exp(\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i)$, то есть суммировать по всем возможным контекстам, по всем словам из словаря, которых сотни тысяч... что же делать?

Миколов с соавторами в работе [123] предложили для решения такой сложной задачи оптимизации специальный метод под названием *отрицательное сэмплирование* (negative sampling). Идея проста: вместо того чтобы считать полную сумму $\sum_{c'} \exp(\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i)$, давайте выберем несколько ее элементов случайным образом в качестве отрицательных примеров и обновим только их, то есть заменим сумму по всем c' на гораздо более маленькую сумму $\sum_{c'\in D'} \exp(\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i)$, где D' — случайно выбранное небольшое подмножество отрицательных примеров.

Почему это вообще может работать? По сути отрицательное сэмплирование — это тоже попытка записать общее правдоподобие, но правдоподобие немножко другого события. Рассмотрим пару (i,c) из слова i и контекста c; тогда наша цель — максимизировать правдоподобие того события, что пара (i,c) появилась из данных, то есть вероятность $p((i,c) \in D;\boldsymbol{\theta})$, которую мы параметризовали некоторым вектором параметров $\boldsymbol{\theta}$. На самом деле таких пар у нас много, и мы хотели бы найти:

$$\arg\max_{\boldsymbol{\theta}} \prod_{(i,c)\in D} p((i,c) \in D;\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{(i,c)\in D} \log p((i,c) \in D;\boldsymbol{\theta}).$$

Теперь давайте параметризуем $p((i,c) \in D;\boldsymbol{\theta})$ через все тот же softmax. Здесь возможных исхода всего два, и softmax превращается в логистический сигмоид $\sigma(x) = \frac{1}{1+\exp(-x)}$:

$$p((i,c) \in D;\boldsymbol{\theta}) = \frac{1}{1 + \exp\left(-\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right)}.$$

Теперь мы хотим максимизировать логарифм общего правдоподобия данных:

$$\arg\max_{\boldsymbol{\theta}} \sum_{(i,c)\in D} \log p((i,c)\in D;\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_{(i,c)\in D} \log \frac{1}{1+\exp\left(-\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right)}.$$

Вот такая задача; как ее решить? Да очень просто: оптимальное значение каждого логарифма будет достигнуто, если $\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i$ окажется как можно больше; никаких других ограничений нет, так что давайте установим все векторы $\tilde{\boldsymbol{w}}_c$ и $\boldsymbol{w}_i$ равными друг другу и с нормой побольше, и добьемся тем самым оптимального правдоподобия, практически равного единице. Стоп, что-то здесь не так...

Подвох в том, что у нас де-факто получился набор данных для бинарной классификации (из данных пара (i,c) или нет), в котором есть только положительные примеры (собственно весь набор данных D) и совсем нет отрицательных! Поэтому ничего удивительного, что оптимальный классификатор состоит в том, чтобы всегда отвечать «да».

Именно эту проблему можно решить отрицательным сэмплированием. Давайте соберем немножко отрицательных примеров, просто выбирая случайные слова и контексты к ним, которых нет в данных; предположение здесь в том, что если мы будем сочетать случайные слова, разумный текст получится с очень маленькой вероятностью, как в борхесовской «Вавилонской библиотеке».

Если мы наберем из таких окон со случайными словами «отрицательное» множество D', то задача максимизации правдоподобия будет выглядеть так:

$$\arg\max_{\boldsymbol{\theta}} \prod_{(i,c)\in D} p((i,c)\in D;\boldsymbol{\theta}) \prod_{(i',c')\in D'} p((i',c')\notin D;\boldsymbol{\theta}).$$

Сделаем несколько арифметических преобразований, выделив все, что касается одного примера $(i,c)\in D$:

$$\arg\max_{\boldsymbol{\theta}} \prod_{(i,c)\in D} p((i,c)\in D;\boldsymbol{\theta}) \prod_{(i',c')\in D'} \big(1-p((i',c')\in D;\boldsymbol{\theta})\big) =$$

$$= \arg\max_{\boldsymbol{\theta}} \left[\sum_{(i,c)\in D} \log p((i,c)\in D;\boldsymbol{\theta}) + \sum_{(i',c')\in D'} \log\big(1-p((i',c')\in D;\boldsymbol{\theta})\big)\right] =$$

$$= \arg\max_{\boldsymbol{\theta}} \sum_{(i,c)\in D} \left[\log \frac{1}{1+\exp\left(-\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right)} + \sum_{(i,c')\in D'} \log \frac{1}{1+\exp\left(\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right)}\right] =$$

$$= \arg\max_{\boldsymbol{\theta}} \sum_{(i,c)\in D} \left[\log \sigma\left(\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right) + \sum_{(i,c')\in D'} \log \sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right)\right].$$

Это и есть основная формула отрицательного сэмплирования из [123]: для каждого окна мы выбираем несколько случайных отрицательных примеров D' и делаем шаг градиентного спуска для функции потерь:

$$-\log\sigma\left(\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right) - \sum_{(i,c')\in D'} \log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right).$$

То же самое происходит и для CBOW-варианта word2vec; не будем, пожалуй, повторять еще раз практически идентичный вывод, но читатель может убедиться самостоятельно, что все работает точно так же.

Прежде чем переходить к практике — еще один интересный взгляд на word2vec. В работе [323] дается интепретация модели word2vec как матричного разложения. Рассмотрим опять функцию потерь, которая оптимизируется в модели word2vec:

$$\ell = \sum_{(i,c)\in D} \left(\log\sigma\left(\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right) + k\mathbb{E}_{c'}\left[\log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right)\right]\right),$$

где k — это число примеров отрицательного сэмплирования. Давайте перепишем суммы немножко по-другому, собирая вместе сумму по каждой паре (i,c); будем обозначать через $n_{i,c}$ число раз, которое она встретится в корпусе, через n_i — число вхождений слова i, через n_c — контекста c. Получится

$$\ell = \sum_i \sum_c \left(n_{i,c}\log\sigma\left(\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right) + kn_{i,c}\mathbb{E}_{c'}\left[\log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right)\right]\right),$$

и вторую часть суммы можно переписать так:

$$\sum_i \sum_c kn_{i,c}\mathbb{E}_{c'}\left[\log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right)\right] = \sum_i kn_i\mathbb{E}_{c'}\left[\log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right)\right],$$

потому что выражение под суммой не зависит от c. Раньше мы оценивали ожидание через сэмплы, а теперь давайте выпишем его явно и полностью:

$$\mathbb{E}_{c'}\left[\log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right)\right] = \sum_{c'} \frac{n_{c'}}{|D|}\log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right) =$$
$$= \frac{n_c}{|D|}\log\sigma\left(-\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right) + \sum_{c'\neq c} \frac{n_{c'}}{|D|}\log\sigma\left(-\tilde{\boldsymbol{w}}_{c'}^\top \boldsymbol{w}_i\right).$$

А значит, если собрать из сумм все, что касается одной отдельно взятой пары (i,c), мы получим:

$$\ell_{i,c} = n_{i,c}\log\sigma\left(\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right) + kn_i\frac{n_c}{|D|}\log\sigma\left(-\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i\right).$$

Чтобы это оптимизировать, обозначим $x = \tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i$ и продифференцируем по x обе части равенства:

$$\frac{\partial \ell_{i,c}}{\partial x} = n_{i,c}\sigma(-x) - kn_i\frac{n_c}{|D|}\sigma(x).$$

Эту производную можно теперь приравнять к нулю; получится квадратное уравнение относительно e^x:

$$e^{2x} - \left(\frac{n_{i,c}}{kn_i\frac{n_c}{|D|}} - 1\right)e^x - \frac{n_{i,c}}{kn_i\frac{n_c}{|D|}} = 0.$$

У него два корня, из которых -1 нам не подходит, потому что экспонента отрицательной быть не может:

$$e^x = \frac{n_{i,c}}{kn_i\frac{n_c}{|D|}} = \frac{1}{k}\frac{n_{i,c}|D|}{n_i n_c},$$

и получается, что для того чтобы оптимизировать исходную функцию правдоподобия, с точным представлением ожидания отрицательных примеров, достаточно подобрать такие $\tilde{\boldsymbol{w}}_c$ для контекстов и $\boldsymbol{w}_i$ для слов, чтобы выполнялось следующее:

$$\tilde{\boldsymbol{w}}_c^\top \boldsymbol{w}_i = \log\left(\frac{n_{i,c}|D|}{n_i n_c}\right) - \log k.$$

Кстати, $\log\left(\frac{n_{i,c}|D|}{n_i n_c}\right)$ — это хорошо известная в теории информации и машинном обучении величина, *поточечная взаимная информация* (pointwise mutual information, PMI). А решение задачи поиска таких $\tilde{\boldsymbol{w}}_c$ и $\boldsymbol{w}_i$ — это, с точностью до аддитивной константы, и есть сингулярное разложение матрицы поточечных взаимных информаций!

Конечно, подобрать $\tilde{\boldsymbol{w}}_c$ и $\boldsymbol{w}_i$ так, чтобы все эти равенства выполнялись для всех i и c точно, не получится (вернее, для этого нужно было бы искать векторы слишком большой размерности, бесполезные на практике), но приближенное решение — это как раз классическая вычислительная задача сингулярного разложения, которую люди умеют решать давно и очень успешно. Точнее, говоря, разложение получается взвешенное, потому что более часто встречающиеся пары (i,c) имеют больший вес в целевой функции. Но это делу не мешает, все равно можно использовать стандартные библиотеки и раскладывать матрицу PMI или ее неотрицательного варианта, в котором элементами являются $\max(0, \mathrm{PMI}(i,c))$. Эксперименты в работе [323] показывают, что такой метод обучения в одних задачах работает лучше обычного word2vec, а в других ему проигрывает; но в любом случае это любопытный интересный новый взгляд на word2vec, с другой стороны иллюстрирующий происходящее.

7.3. Русскоязычный word2vec на практике

> Для этого Тютчев избирает знаменательный путь: он дает на русской почве аналогию приемов немецкого стихотворения, оставаясь все время, однако, верным своей лексической традиции.
>
> *Ю. Тынянов. Тютчев и Гейне*

Итак, в теории все более или менее работает, а что же с практикой? Реализовать word2vec на такой удобной библиотеке, как TensorFlow, большого труда не составляет; пример такой реализации можно найти даже в официальной документации TensorFlow[1]. Однако на практике, если вам нужно использовать базовую модель word2vec, а не расширять ее в ту или иную сторону, мы не рекомендуем реализовывать word2vec самостоятельно. Дело в том, что хотя word2vec и другие модели распределенных представлений слов очень легко реализовать на Keras, TensorFlow или другой современной библиотеке для обучения глубоких сетей, эта реализация будет далеко не столь эффективной, как уже существующие специализированные реализации. Да и сами алгоритмы для обучения word2vec претерпели существенную эволюцию; в более поздних работах [137, 338, 339, 370] появился ряд более эффективных и/или более стабильных алгоритмов для обучения представлений слов, которые фактически делают то же самое, оптимизируют те же целевые функции, но лучше. А суть модели такова, что ее хочется запустить на как можно большем объеме данных, и собрать его совсем несложно, текстов доступно очень много... Поэтому здесь мы рекомендуем велосипедов не изобретать, и сами тоже не будем приводить код для модели word2vec, а будем предполагать, что нам просто нужно правильно настроить параметры какой-нибудь стандартной реализации.

Для этого мы воспользуемся библиотекой Gensim [446], которая в последние годы стала одной из стандартных библиотек не только для word2vec (поддержка word2vec как раз была добавлена относительно недавно), а для современных моделей обработки текста вообще; Gensim очень эффективно реализует ряд моделей для тематического моделирования, о которых мы немного говорили выше: латентно-семантический анализ, латентное размещение Дирихле и иерархические процессы Дирихле. Собственно обучение моделей word2vec было портировано в Gensim из исходного кода, выпущенного компанией Google[2]. Кстати, очень рекомендуем прочитать посты создателя Gensim Радима Рехурека [445] о том, как он портировал word2vec в Python. Получилась увлекательная история о том, как на практике добиться топ-производительности в *Python*:

- первая наивная версия работала в 1000 раз медленнее исходного кода на C;
- после *NumPy*-оптимизаций получалось все равно в 20 раз медленнее;

[1] https://www.tensorflow.org/versions/r0.9/tutorials/word2vec/index.html.

[2] https://code.google.com/archive/p/word2vec/.

- после того как основные «бутылочные горлышки» в коде были оптимизированы с помощью *Cython* [60], базовую *NumPy*-версию удалось ускорить более чем в 20 раз, и *Python*-реализация уже немного превосходила по скорости код word2vec от Google (речь пока об однопоточной версии);
- векторизация некоторых операций и прямые обращения к базовой библиотеке алгоритмов линейной алгебры BLAS [546] позволили ускорить все это еще в три с лишним раза;
- а затем разумная параллелизация помогла ускорить код почти линейно от числа потоков.

В результате сложилась достаточно редкая и очень интересная ситуация: многопоточный код обучения word2vec в Gensim работает зачастую даже *быстрее*, чем реализации обучения word2vec-модели в стандартных для обучения глубоких сетей библиотеках TensorFlow или Theano, безо всякого GPU. Кроме того, Gensim предоставляет некоторые другие важные и удобные инструменты обработки больших текстовых корпусов, которыми мы тоже воспользуемся в примере ниже.

Здесь мы будем обучать word2vec на общедоступном и довольно большом корпусе текстов: корпусе русскоязычной «Википедии». Текущую версию корпуса можно скачать с сервера Wikimedia, хранящего полные архивы (дампы) разных проектов Wikimedia foundation[1].

Для экспериментов с моделями word2vec мы использовали состояние русскоязычной «Википедии» от 20 октября 2016 года: bz2-архив общим объемом 2,9 Гбайт, содержащий 1 107 149 текстов статей в формате XML. Это не самый большой корпус на свете, но он уже вполне разумного размера, покрывает множество разных тем, и мы вполне можем надеяться, что результаты экспериментов окажутся достаточно интересными и показательными.

Первое дело, которое неизбежно является частью любого проекта по обработке текстов, — это предобработка. Часто довольно простая техническая предобработка — привести все тексты к единому формату, разделить на слова и т. п. — занимает не меньше времени и сил, чем собственно построение содержательной модели. Кстати, процесс разделения потока символов на токены — слова, которые будут потом обрабатываться алгоритмом как единое целое, называется *токенизацией*. Во многих статьях на тему обработки текстов токенизация пропускается за очевидностью, потому что для английского языка она обычно действительно делается довольно просто: нужно разделить текст по пробелам, убрать или выделить в отдельные слова знаки препинания, перевести в нижний регистр, вот и все.

Дело может оказаться более сложным для языков с богатой морфологией, например русского; здесь все зависит от того, хотите ли вы перед использованием моделей проводить *лемматизацию*, то есть переводить слова в начальные формы, чтобы из предложения «мама с дочкой мыли раму» получалось «мама с дочка мыть рама».

[1] https://dumps.wikimedia.org/

Лемматизация сильно сократит словарь и увеличит «связность» текстов, поэтому при использовании методов, которые не должны понимать смысл синтаксических конструкций, а просто используют тексты как «мешки слов» (bag of words), мы рекомендуем делать лемматизацию. Но ниже, в обучении word2vec и других моделей, мы не будем лемматизировать тексты по нескольким причинам: во-первых, для простоты, во-вторых, чтобы увидеть некоторые связанные с синтаксисом эффекты, а в-третьих, потому, что данных у нас будет достаточно много, чтобы обучить разумные модели и на исходных текстах.

Отметим здесь, что существует и еще более сложный случай, возникающий в анализе китайского языка и некоторых похожих на него: текст на китайском языке выглядит как последовательность иероглифов, никаких пробелов там нет, но при этом одно слово может записываться несколькими иероглифами. Поэтому для китайского языка токенизация — это отдельное сложное дело: с ходу вообще непонятно, как разделить текст на слова (мы упоминали в разделе 7.1 задачу пословной сегментации), и для этого нужно обучать отдельные модели, которые мы в этой книге рассматривать все-таки не будем.

При обработке корпуса «Википедии» мог бы возникнуть как раз такой случай, потому что он оформлен в специальном XML-формате, который нужно было бы разбирать. К счастью, здесь Gensim нам поможет: в нем есть специальный класс для загрузки и анализа именно дампов «Википедии», так что вот весь код, который нам нужно написать для этого:

```
from gensim.corpora.wikicorpus import WikiCorpus
wiki = WikiCorpus('ruwiki-20161020-pages-articles-multistream.xml.bz2',
    dictionary=False)
```

Корпус при этом в память целиком не загружается, он остается на диске, и дорогостоящая процедура построения словаря благодаря ключу `dictionary=False` тоже не делается, а для доступа к текстам можно пользоваться генератором `wiki.get_texts()`:

```
for text in wiki.get_texts():
```

Но это еще не вся магия Gensim. Следующий этап предобработки, который почти всегда разумно сделать в анализе текстов — выделение *биграмм* (а также, возможно, триграмм и более). Хотя биграммами часто называют просто любые пары рядом стоящих слов, в этом контексте биграмма — это пара слов, которые мы объединяем вместе и рассматриваем как единое целое, считаем одним токеном. Это нужно потому, что в естественных языках часто устойчивые выражения не являются суммой составляющих их слов, а то и вовсе не имеют к ним никакого отношения. Например, словосочетания «может быть» или «в течение» можно понять буквально, по словам, но в реальности они уже давно никем так не рассматриваются, это единые устоявшиеся конструкции[1]. То же относится к словосочетаниям,

[1] Сразу предупредим, что мы не лингвисты, но если немного экстраполировать наши любительские познания в лингвистике, то представляется, что в естественном, устном языке такие устойчивые

обозначающим единый объект, например «машинное обучение», их тоже логично было бы рассматривать как единое целое.

Конечно, еще лучше выделять триграммы, тетраграммы и т. д. без ограничений, но число n-грамм быстро начинает расти экспоненциально, и в память они помещаться перестают, сколько бы ее ни было.

Существует целый ряд алгоритмов для выделения биграмм, но общая их суть примерно одинакова: биграмма — это пара слов (w_1,w_2), которые вместе встречаются аномально часто, то есть совместная вероятность их появления $p(w_1,w_2)$ в тексте существенно выше, чем $p(w_1)p(w_2)$, та частота, которую мы бы наблюдали, если бы эти слова появлялись независимо друг от друга.

Что такое «существенно» и как выделить такие случаи, учитывая, что матрицу совместной встречаемости всех-всех слов в гигантском корпусе мы скорее всего построить не можем, и т. д., — это как раз те вопросы, в которых конкретные алгоритмы выделения биграмм различаются. Но они не являются предметом нашей книги, так что мы сейчас не будем на них подробно останавливаться, а просто скажем, что в Gensim реализован такой алгоритм, и запустить его очень просто:

```
from gensim.models.phrases import Phrases,Phraser
bigram = Phrases(wiki.get_texts())
bigram_transformer = Phraser(bigram)
```

Этот код на большом корпусе будет выполняться довольно долго, но зато в результате получится словарь биграмм, который можно использовать, применяя к потоку текстов `bigram_transformer`. Вот так можно построить генератор для текстов с биграммами:

```
def text_generator_bigram():
    for text in wiki.get_texts():
        yield bigram_transformer[ [ word.decode('utf-8') for word in text ] ]
```

Некоторые конкретные примеры показаны в табл. 7.1; для каждой биграммы Gensim подсчитывает ее общее число вхождений, а также оценку того, насколько «слишком часто» эти слова встречаются вместе, в виде

$$\text{score}(i,j) = \frac{n_{i,j} - \delta}{n_i n_j},$$

где $n_{i,j}$ — число вхождений биграммы, n_i и n_j — число вхождений слов i и j по отдельности, а δ — небольшая константа, нужная для того, чтобы отсечь «длинный хвост» биграмм из редких слов. Мы выписали некоторые частые биграммы, выбросив технические вроде *примечания_ссылки* или биграммы со словом «категория», которые получались из структуры «Википедии». Получается вполне разумно.

словосочетания уже давно выделились в отдельные «слова» и начали бы видоизменяться совершенно отдельно от своих былых составных частей, и только традиции письменности мешают им стать единым словом.

Таблица 7.1. Примеры биграмм из русскоязычной «Википедии»

Биграмма	Число вхождений	score
во_время	261 546	11,38
см_также	204 070	13,89
том_числе	133 648	58,26
игроки_фк	127 001	93,98
при_этом	121 743	12,86
населенные_пункты	117 164	192,41
великой_отечественной	116 947	146,15
российской_федерации	114 249	92,63
отечественной_войны	114 129	50,30
официальный_сайт	100 211	123,99
настоящее_время	97 224	29,33
советского_союза	86 693	122,54

А чтобы обучить набор триграмм, то есть устойчивых последовательностей из трех слов (например, «по моему мнению» или «большой адронный коллайдер») нужно просто обучить биграммы два раза подряд: тогда некоторые из них «подтянут» к себе еще одно слово и станут триграммами (а в отдельных случаях, возможно, даже обучатся последовательности из четырех слов). Для этого можно использовать тот же класс `Phrases`:

```
trigram = Phrases(text_generator_bigram())
trigram_transformer = Phraser(trigram)
def text_generator_trigram():
    for text in wiki.get_texts():
        yield trigram_transformer[ bigram_transformer[
            [ word.decode('utf-8') for word in text ] ] ]
```

Примеры для случая русскоязычной «Википедии» показаны в табл. 7.2; и снова примеры довольно логичные, просто нужно понимать, что некоторые стандартные для энциклопедий обороты вроде «по данным водного реестра России» будут встречаться столько раз, сколько в России рек и озер, то есть много[1]. Заметьте, кстати, что если вы повторите этот код, то большинство триграмм будут встречаться в двух эквивалентных вариантах: *второй_мировой_войны* могло получиться слиянием *второй_мировой* и *войны* или *второй* и *мировой_войны*, и для Gensim это будут, конечно, разные пары токенов. Но для нас это несущественно, потому что использовать статистику и оценки триграмм мы не будем, все, что нам нужно, — это список би- и триграмм, который можно использовать для преобразования входного текста.

[1] Самая загадочная триграмма китт_пик_spacewatch тоже из этой категории: речь идет о Национальной обсерватории Китт-Пик, находящейся в Аризоне. В рамках проекта Spacewatch обсерватория Китт-Пик открыла невероятное множество разных астероидов, каждый из которых, конечно, ничем особо не примечателен, но в «Википедии» есть статьи вроде «Список астероидов (120501–120600)».

Таблица 7.2. Примеры триграмм из русскоязычной «Википедии»

Триграмма	Число вхождений	score
великой_отечественной_войны	41 046	846,51
герой_советского_союза	40 443	1160,52
второй_мировой_войны	39 405	627,38
летних_олимпийских_играх	39 282	11478,73
верховного_совета_ссср	24 334	16040,93
первой_мировой_войны	24 075	576,75
китт_пик_spacewatch	23 989	513738,05
указом_президиума_верховного_совета	23 229	240769,59
по_данным_водного	23 036	3569,17
данным_водного_реестра_россии	22 955	388516,09
данные_водного_реестра	22 599	31277,18
до_сих_пор	22 329	15297,65
по_данным_переписи	19 246	194,37
по_данным_системы	18 377	82,25

Теперь все готово: давайте обучать векторы word2vec на последовательности триграмм из корпуса русскоязычной «Википедии». В Gensim это, опять же, делается буквально за пару строчек кода:

```
from gensim.models.word2vec import Word2Vec
model = Word2Vec(size=100, window=7, min_count=10, workers=10)
model.build_vocab( text_generator_trigram() )
model.train( text_generator_trigram() )
```

Размерность вектора задается параметром `size`, `window` определяет размер окна локального контекста, `min_count` означает, что слова с частотой встречаемости меньше этой в корпусе мы рассматривать и обучать для них векторы не будем, а `workers` — это как раз параметр, связанный с параллелизацией, то, во сколько потоков мы будем обучать модель. Обратите внимание, что долгим процессом здесь является не только собственно обучение, `model.train`, но и создание словаря `model.build_vocab`: чтобы построить словарь, тоже нужно пройтись по всему корпусу и посчитать вхождения всех слов, а строить его на лету не получится, если хочется хорошо соптимизировать `model.train`.

Однако же после всего этого получается обученная модель word2vec, на которой можно увидеть много разных интересных эффектов. Мы обучили несколько моделей разной размерности, чтобы продемонстрировать, какая между ними разница; время обучения показано в табл. 7.3. Обратите внимание, что в нашем эксперименте при росте размерности базового вектора обучение вовсе не растет линейно по времени. В данном случае, по всей видимости, «бутылочным горлышком» был не сам процесс обучения модели, а параллельное чтение датасета с диска: хотя наш архив «Википедии» лежал на достаточно быстром SSD-диске, при обучении

Таблица 7.3. Время работы обучения моделей word2vec в Gensim на корпусе русскоязычной «Википедии»

Модель	Размерность	Время обучения
Подсчет биграмм		34 мин
Подсчет триграмм		47 мин
Создание словаря		52 мин
Обучение word2vec	50	54 мин
Обучение word2vec	100	53 мин
Обучение word2vec	200	53 мин
Обучение word2vec	500	57 мин

в 10 процессов большая часть этих процессов использовали соответствующий поток CPU лишь на 20–30%.

И давайте же посмотрим, что получается! Чтобы получить список ближайших соседей данного токена в модели `model`, достаточно запустить

```
model.most_similar('токен')
```

В табл. 7.4 приведены некоторые примеры ближайших соседей, полученных из модели с 300-мерными векторами вместе со значением похожести (которая в данном случае считается просто как скалярное произведение векторов). Как видите, действительно получается, что модель word2vec в большинстве случаев неплохо поняла семантику, пользуясь всего лишь локальными контекстами слов.

А в табл. 7.5 показаны несколько примеров линейных соотношений в семантическом пространстве векторов: арифметика векторов «король + женщина - мужчина» должна, если повезет, означать, что мы ищем слово, которое так же относится к «королю», как «женщина» относится к «мужчине». Пример про «king + woman − man» — самый классический в англоязычной литературе, так что хорошо, что у нас он тоже удался. В примере с «математиком» мы видим, что модели присущ некоторый сексизм: женщина-математик — это, оказывается, филолог.[1] Со столицами тоже работает неплохо: французская Москва оказалась действительно Парижем. А вот с авторами и их произведениями уже не так все однозначно; мы привели хороший пример, но его пришлось поискать, хотя в среднем истина обычно где-то рядом.

[1] Это свойство легко объяснить: женщин-филологов больше, чем женщин-математиков, в том числе и в «Википедии», и логично, что они будут попадаться в тех же контекстах, соответствующих общему понятию «ученого». Однако любопытно, что на это свойство word2vec обнаруживать и подкреплять стереотипы уже обратили внимание исследователи, которые пытаются убрать сексистский или расистский уклон из моделей машинного обучения [347]. Существует целое направление под названием Fairness in ML, которое занимается тем, что декоррелирует результаты моделей с атрибутами, дискриминировать по которым нельзя. На первый взгляд это может показаться странной деятельностью, но речь уже сейчас часто идет о системах, принимающих решения о вашей жизни: согласитесь, в наше время странно было бы увидеть отказ в кредите на единственном основании «неправильной» национальности.

Таблица 7.4. Примеры ближайших соседей в модели word2vec

машина_времени		**машинное_обучение**	
наутилус_помпилиус	0,869	криптография	0,890
король_шут	0,863	объектно_ориентированное	0,881
сплин	0,850	математическая_логика	0,880
агата_кристи	0,848	разработка_программного_обеспечения	0,879
мумий_тролль	0,844	параллельные_вычисления	0,878
андрей_макаревич	0,839	обработка_естественного_языка	0,871
ляпис_трубецкой	0,836	теория_алгоритмов	0,870
борис_гребенщиков	0,835	информационная_безопасность	0,869
чайф	0,835	математическая_статистика	0,865
ва_банкъ	0,829	обработка_сигналов	0,864
микеланджело		**микеланджело_антониони**	
караваджо	0,867	федерико_феллини	0,913
джотто	0,861	стэнли_кубрик	0,912
вазари	0,842	бернардо_бертолуччи	0,904
тициана	0,837	франческо_рози	0,899
боттичелли	0,834	марио_моничелли	0,898
мазаччо	0,829	этторе_скола	0,896
бернини	0,822	франсуа_трюффо	0,896
дюрера	0,821	роман_полански	0,896
леонардо_да_винчи	0,821	пролетая_над_гнездом_кукушки	0,891
рубенса	0,816	секреты_лос_анджелеса	0,890
вторая_мировая_война		**великая_отечественная_война**	
первая_мировая_война	0,823	началом_великой_отечественной_войны	0,797
корейская_война	0,720	великую_отечественную_войну	0,790
люфтваффе	0,697	первые_дни_войны	0,755
война_во_вьетнаме	0,685	под_ленинградом	0,751
межвоенный_период	0,678	сталинграде	0,743
холодная_война	0,674	годы_великой_отечественной_войны	0,740
третьего_рейха	0,669	ленинградского_фронта	0,728
третий_рейх	0,666	ркка	0,727
нацистская_германия	0,661	великой_отечественной	0,726
стран_оси	0,661	рабоче_крестьянской_красной_армии	0,723
александр_македонский		**леонид_ильич_брежнев**	
митридат	0,828	леонид_брежнев	0,850
александра_великого	0,817	председатель_совета_министров_ссср	0,834
деметрий	0,815	иосиф_виссарионович_сталин	0,816
александра_македонского	0,812	генеральный_секретарь_цк_кпсс	0,805
сузы	0,809	никита_сергеевич_хрущев	0,775
марк_антоний	0,796	николай_викторович_подгорный	0,773
гай_юлий_цезарь	0,795	председатель_совета_народных_комиссаров	0,772
дарий	0,795	член_политбюро_цк_кпсс	0,770
селевка	0,792	председатель_кгб_ссср	0,770
римский_император	0,791	вячеслав_михайлович_молотов	0,765

Таблица 7.5. Примеры линейных соотношений в модели word2vec

король + женщина – мужчина = …		**математик + женщина – мужчина = …**	
королева	0.624	филолог	0.667
империя	0.562	переводчица	0.666
принцесса	0.552	доктор_философии	0.660
правительница	0.532	доктор_филологических_наук	0.656
королевская_семья	0.531	лингвист	0.655
короля	0.510	социолог	0.652
аристократия	0.510	аушра_аугустинавичюте	0.651
инквизиция	0.503	доктор_филологических_наук_профессор	0.650
императрица	0.503	кандидат_филологических_наук	0.649
королева_елизавета	0.500	поэтесса	0.648
москва + франция – россия = …		**укрощение_строптивой + гоголь – шекспир = …**	
париж	0.566	ночь_перед_рождеством	0.791
фр	0.530	за_двумя_зайцами	0.791
париж_франция	0.493	ревизор_гоголя	0.787
жан	0.489	живой_труп	0.784
французский	0.486	без_вины_виноватые_островского	0.783
брюссель	0.486	вишневый_сад	0.779
франсуа	0.485	волки_овцы_островского	0.778
анри	0.485	александринский_театр	0.778
женева	0.481	братья_карамазовы	0.777
paris	0.480	мертвые_души	0.776

7.4. GloVe: раскладываем матрицу правильно

> У Тютчева... внешняя художественная форма не является надетой на мысль, как перчатка на руку, а срослась с нею, как покров кожи с телом, сотворена вместе и одновременно, одним процессом: это сама плоть мысли.
>
> *И. С. Аксаков*

Самой популярной современной альтернативой word2vec являются модели GloVe (от слов *global vectors*) [417]. Основная идея GloVe состоит в том, чтобы сочетать сильные стороны двух подходов-предшественников, одновременно избегая их недостатков:

- для методов, основанных на матрице встречаемости, таких как LSA и его современные варианты, нужно как-то избежать проблем, связанных с сильной неравномерностью совместной встречаемости;

- а локальные методы, такие как word2vec, хотелось бы все-таки переложить на язык матрицы совместной встречаемости, потому что в word2vec при обучении приходится проходить все окна локального контекста подряд, а матрица встречаемости могла бы «схлопывать» повторяющиеся окна и вообще сильно сжимать данные, просто подсчитывая общие статистики.

Модель GloVe можно считать продолжательницей дела LSA, она тоже пытается приблизить матрицу встречаемости слов, но делает это весьма интересным и не совсем обычным образом. Чтобы объяснить, в чем новизна, введем сначала некоторые обозначения: пусть $X \in \mathbb{R}^{V\times V}$ — это матрица совместной встречаемости слов, то есть X_{ij} показывает то, сколько раз в нашем корпусе слово i встретилось вместе со словом j, а $X_i = \sum_j X_{ij}$ — общее число раз, которое какое-то другое слово встречается в контексте слова i. Тогда легко превратить эти счетчики в оценки вероятностей; давайте обозначим

$$p_{ij} = p(j \mid i) = \frac{X_{ij}}{X_i} = \frac{X_{ij}}{\sum_k X_{ik}},$$

то есть p_{ij} — это вероятность того, что слово j встретится в контексте слова i.

Пока ничего удивительного, эти вероятности можно попытаться приблизить, раскладывая матрицу таких вероятностей, и получится просто еще один вариант LSA. Но GloVe приближает не сами вероятности, а их *отношения*. Дело в том, что когда мы смотрим на сами вероятности $p_{ij} = p(j \mid i)$, их значения не очень понятно как интерпретировать и как сравнивать между собой. Но вот вероятности p_{ik} и p_{jk} *для одного и того же слова* k уже вполне сравнимы: одна показывает, как часто в контексте слова k встретится слово i, а другая — как часто встретится слово j; понятно, что если p_{ik} существенно больше, чем p_{jk}, то слово i теснее связано со словом k, чем слово j.

Давайте приведем конкретный пример: подсчитаем такие статистики для некоторых слов из русскоязычной «Википедии». В том архиве «Википедии», который мы взяли для экспериментов, было около 900 тысяч статей. Заметим, кстати, к вопросу о размере словарей, что эти 900 тысяч статей, которые предположительно написаны адекватным русским языком, без огромного количества опечаток и окказионализмов, содержат 3 700 525 разных слов! Конечно, в эти миллионы вошло большое количество разных форм одного и того же слова, но с ними ведь система обработки текстов тоже должна уметь что-то делать... но об этом позже.

А пока давайте посмотрим на конкретный пример, приведенный в табл. 7.6. Мы подсчитали встречаемость слов в русскоязычной «Википедии» в локальных контекстах друг у друга с окном ширины 8, а затем вычислили статистику совместной встречаемости со словами «клуб» и «команда» для слов «футбол», «хоккей», «гольф» и «корабль»[1]. Интуитивно понятно, каких результатов мы ожидаем:

[1] Замечание для въедливых читателей: в этом эксперименте мы не проводили никакой лемматизации, поэтому абсолютные цифры довольно низкие; мы подсчитывали только случаи, в которых слова

Таблица 7.6. Примеры частоты встречаемости слов в русскоязычной «Википедии»

Слово k	Число вхождений			Вероятности $p(k \mid \ldots)$, $\times 10^{-4}$		Отношение $\frac{p(k\mid\text{клуб})}{p(k\mid\text{команда})}$
	Всего	Вместе с:				
		клуб	команда	клуб	команда	
футбол	29 988	54	34	18,0	11,3	1,588
хоккей	10 957	16	7	6,39	14,6	2,286
гольф	2721	11	1	40,4	3,68	11,0
корабль	100 127	0	30	0,0	3,00	0,0

- в футболе и хоккее слова «клуб» и «команда» практически взаимозаменяемы;
- в гольфе бывают гольф-клубы, но команды гольфистов встречаются редко;
- на корабле команда есть обязательно, а вот с клубами сложнее.

Если мы посмотрим на абсолютные цифры совместной встречаемости, они нам ожидаемо ничего хорошего не скажут: 16 раз, которые встретились «хоккей» и «клуб», похожи на 11 раз у «гольфа» и «клуба» гораздо больше, чем на 54 раза у «футбола». Вероятности p_{ij} будут уже более информативными, но при этом очень неровными: получается, что соотношение у «корабля» и «команды» должно быть примерно таким же, как у «гольфа» и «команды», хотя в первом случае это много, а во втором — мало. Поэтому обучать их все равно не слишком приятно. А отношения вероятностей выглядят наиболее разумно: видно, что «клуб» и «команда» употребляются в хоккейных и футбольных контекстах примерно одинаково (в хоккее слово «команда» и впрямь употребляется несколько реже, чем в футболе), а значения отношения $\frac{p(k\mid\text{клуб})}{p(k\mid\text{команда})}$ для «гольфа» и «корабля» совершенно очевидно существенно больше и меньше единицы соответственно.

Примерно это и хочет выразить модель GloVe, поэтому моделирует не сами вероятности p_{ij}, а их отношения, обучая функцию

$$F(\boldsymbol{w}_i, \boldsymbol{w}_j; \tilde{\boldsymbol{w}}_k) = \frac{p_{ij}}{p_{jk}},$$

где $\boldsymbol{w}_i$ и $\boldsymbol{w}_j$ — это векторы слов i и j в пространстве $\mathbb{R}^d$, а $\tilde{\boldsymbol{w}}_k$ — это так называемые *векторы контекста*, которые должны отразить тот факт, что соотношение между словами i и j мы сейчас приближаем не вообще, а именно в контексте слова k.

Осталось только понять, что взять в качестве функции F. Теоретически F могла бы быть очень сложной функцией: мы могли бы, например, подать конкатенацию векторов $\boldsymbol{w}_i$, $\boldsymbol{w}_j$ и $\tilde{\boldsymbol{w}}_k$ на вход глубокой нейронной сети, которая сделала бы с ними что-нибудь ужасное. Но наша конечная цель вовсе не в том, чтобы хорошо приблизить матрицу совместной встречаемости, а в том, чтобы получить хорошие

встречаются друг рядом с другом именно в такой начальной форме, «футбольный клуб» или «команда корабля» не подходят.

векторы! Это такие векторы, между которыми есть простые и понятные взаимоотношения, вроде король − мужчина + женщина = королева. Поэтому создатели GloVe сразу резко ограничили набор возможных функций, считая, что отношение вероятностей должно выражаться функцией от одного аргумента, скалярного произведения разницы между $\boldsymbol{w}_i$ и $\boldsymbol{w}_j$ на $\tilde{\boldsymbol{w}}_k$:

$$F(\left(\boldsymbol{w}_i - \boldsymbol{w}_j\right)^\top \tilde{\boldsymbol{w}}_k) = \frac{p_{ij}}{p_{jk}}.$$

Благодаря этому ограничению GloVe будет пытаться обучить именно линейные соотношения между векторами, представляющими слова.

Осталось только отметить, что функция F должна еще обладать некоторой внутренней симметрией, ведь матрица совместной встречаемости слов симметрична, и при переходе от X к $X^\top$ и от $\boldsymbol{w}$ к $\tilde{\boldsymbol{w}}$ ничего меняться не должно; а наша функция пока такой симметрией не обладает. Чтобы ее добавить, давайте введем еще одно предположение: допустим, что F не просто переводит разность между $\boldsymbol{w}_i$ и $\boldsymbol{w}_j$ в отношение соответствующих вероятностей, а и вообще переводит сумму чисел в их произведение, а разность — в отношение[1]:

$$F(\left(\boldsymbol{w}_i - \boldsymbol{w}_j\right)^\top \tilde{\boldsymbol{w}}_k) = \frac{F\left(\boldsymbol{w}_i^\top \tilde{\boldsymbol{w}}_k\right)}{F\left(\boldsymbol{w}_j^\top \tilde{\boldsymbol{w}}_k\right)} = \frac{p_{ij}}{p_{jk}}.$$

Это тоже поможет сделать функцию F «регулярной», постараться выразить еще более простые и глобальные соотношения. И теперь остался один шаг до того, чтобы просто найти функцию F в явном виде: таких отображений не то чтобы очень много, и у нас получается, что F — это просто экспонента:

$$\boldsymbol{w}_i^\top \tilde{\boldsymbol{w}}_k = \log(p_{ik}) = \log(X_{ik}) - \log(X_i).$$

Чтобы векторы слов и векторы контекстов стали симметричны, достаточно теперь добавить еще по свободному члену b_i для слова и $\tilde{b}_j$ для контекста; тогда $\log(X_i)$ можно спрятать в b_i, и получится красивая симметричная модель:

$$\boldsymbol{w}_i^\top \tilde{\boldsymbol{w}}_k + b_i + \tilde{b}_k = \log(X_{ik}).$$

У нее остались только две проблемы:

- во-первых, $\log(X_{ik})$ очень часто будет расходиться, потому что X_{ik}, число совместных появлений двух слов, очень часто равно нулю, и вообще X — матрица весьма разреженная;

[1] Сноска для людей с математическим образованием: это значит, что F является гомоморфизмом из группы $(\mathbb{R}, +)$ в группу $(\mathbb{R}_+, \times)$.

- во-вторых, она по-прежнему взвешивает все X_{ik} одинаково, но относительные частоты двух очень редких слов — дело во многом случайное, а относительные частоты с очень частыми словами — не слишком показательное.

В модели GloVe обе эти проблемы решаются одним махом; для этого мы будем обучать веса $\boldsymbol{w}$ и $\tilde{\boldsymbol{w}}$ не через простую сумму квадратов отклонений, а через взвешенную. Таким образом, в GloVe целевая функция для обучения представлений слов $\boldsymbol{w}_i$ и $\tilde{w}_i$ получается такой:

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \left(\boldsymbol{w}_i^\top \tilde{\boldsymbol{w}}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2,$$

где $X \in \mathbb{R}^{V \times V}$ — это по-прежнему матрица совместной встречаемости слов, $X_i = \sum_j X_{ij}$ — общее число совместных встречаемостей слова i, $\boldsymbol{w} \in \mathbb{R}^d$ — собственно векторное представление слова в евклидовом пространстве размерности d, $\tilde{\boldsymbol{w}} \in \mathbb{R}^d$ — это представление контекста слова в виде вектора той же размерности d, V — размер словаря.

Ну а f — это функция, которая не присваивает частым совместным встречаемостям слишком больших весов, а также обнуляет сомножители, в которых должен был быть логарифм нуля; иначе говоря, нужно, чтобы $f(0) = 0$, а f была неубывающей (чтобы маленькие частоты встречаемости получили маленькие веса), но при этом f не должна слишком быстро возрастать с ростом своего аргумента, чтобы ограничить влияние самых популярных общих слов. Обычно в GloVe используется такая весовая функция:

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^\alpha, & \text{если } x < x_{\max}, \\ 1 & \text{в противном случае.} \end{cases}$$

В работе [417] показано, что векторы GloVe действительно работают не хуже, а часто и лучше word2vec на многих реальных задачах обработки текстов, и с тех пор модель GloVe заняла место одного из двух «стандартных» методов распределенного представления слов.

Эксперименты в работе [417] и последующих работах также показывают очень интересные геометрические эффекты у векторов GloVe: не просто маленькое расстояние между словами в $\mathbb{R}^d$ указывает на семантическую близость, но и семантические соотношения между концепциями превращаются в простые геометрические отношения между векторами слов! А их простые комбинации (обычно сумма или среднее) часто довольно точно отражают смысл соответствующих словосочетаний.

В заключение отметим, что кроме word2vec и GloVe есть, конечно, и другие современные подходы к тому, как реализовать распределенные представления слов.

В дальнейшем в этой главе мы все-таки будем в основном использовать классический word2vec, но давайте сейчас кратко пробежимся и по другим подходам тоже. Мы ограничимся перечислением основных идей, а заинтересованный читатель сможет прочесть подробности по приведенным ниже ссылкам.

Во-первых, ряд достаточно современных работ возвращается к исходной идее латентного семантического анализа и пытается посмотреть на базовую идею разложения матрицы совместной встречаемости слов с более современной точки зрения. В работе [313] авторы строят представления слов с помощью анализа главных компонент (PCA), с той разницей, что используется вариант PCA, минимизирующий так называемое расстояние Хеллингера (Hellinger distance) между полученным распределением и распределением данных. Представления получаются вполне разумные, и Hellinger PCA может служить достойной альтернативой word2vec. А в работе [322] авторы обучают модели, похожие на word2vec, но использующие не локальные окна из последовательности слов, а окна, основанные на дереве зависимостей слов: например, в предложении «мыла раму тряпкой мама» слова «раму» и «тряпкой» будут соседями слова «мыла», а слово «мама» будет соседом вовсе не слова «тряпкой», а слова «мыла»; такие представления имеют несколько иные свойства, чем обычные.

Во-вторых, хотя по результатам экспериментов получается, что модели вроде word2vec и GloVe очень хорошо решают задачу семантической похожести, это все-таки не первые шаги, которые человечество сделало в этом направлении. Уже давно существуют большие базы разного рода семантической информации и проверенных людьми отношений между объектами естественного языка. Например, в одной из крупнейших баз знаний об английском языке, WordNet [154], на данный момент содержится более 117 тысяч так называемых синсетов (synsets), наборов синонимичных слов, не считая прочих семантических отношений между словами. А в крупнейшей семантической базе знаний Freebase [163] содержатся почти *два миллиарда* фактов и отношений между сущностями в виде троек «субъект — предикат — объект»[1], например:

(«Джеффри Хинтон», «родился в», «Уимблдон, Лондон, Великобритания») .

Поэтому вполне естественно, что некоторые работы пытаются дополнить процесс порождения распределенных представлений слов, улучшить базовые модели с помощью этой дополнительной общедоступной информации. Например, в [577] базовая целевая функция word2vec дополняется еще одним слагаемым (можно сказать, регуляризатором), которое вводит дополнительные мягкие ограничения

[1] Мы пишем эти строки в тот момент, когда с базой знаний Freebase происходят серьезные изменения: хотя архивы базы доступны для скачивания и, по всей видимости, останутся доступными и в будущем, база уже дальше не растет, официально проект больше не поддерживается Google, а его новая версия носит название Google Knowledge Graph; к нему есть и программный доступ через общедоступный открытый API на https://developers.google.com/knowledge-graph/.

на векторы слов так, чтобы они как можно лучше выражали уже известные отношения между словами; эксперименты на базе WordNet и базы данных парафразов PPDB [173] показывают значительные улучшения по сравнению с базовыми моделями.

Модель RC-NET [436] расширяет модель *word2vec* информацией от графов знаний, построенных на основе баз знаний вроде Freebase. В RC-NET базовая модель word2vec получает регуляризатор для каждого отношения в базе, пытаясь выразить его линейным соотношением.

Например, модель RC-NET будет стараться сделать так, чтобы предикат родился_в_городе выражался вектором $r_{\text{родился\_в\_городе}}$ (который также обучается в процессе обучения модели), а регуляризатор пытается сделать так, чтобы векторы объектов, связанных этим отношением, отстояли друг от друга как раз на r; иными словами в целевую функцию явно добавляются слагаемые, которые пытаются сделать так, чтобы выполнялось, например:

$$\boldsymbol{w}_{\text{Джеффри\_Хинтон}} - \boldsymbol{w}_{\text{Уимблдон}} \approx r_{\text{родился\_в\_городе}} \approx \boldsymbol{w}_{\text{Эйлер}} - \boldsymbol{w}_{\text{Базель}}.$$

Подобные формы внешних знаний используются и в [384, 485], а работа [42] пытается добавить лингвистические знания о морфологии, синтаксисе и семантике. А в работе [449] предлагается способ сделать примерно то же, что в двух вышеприведенных, но не на этапе обучения модели, а подправляя уже готовые векторы. Это, конечно, с практической точки зрения гораздо удобнее, потому что для таких языков, как английский, уже существуют очень хорошо сделанные готовые распределенные представления слов, обученные на поистине гигантских наборах текстов; переобучать их было бы довольно сложно, и это могло бы свести всю выгоду от дополнительной семантической информации на нет.

Логичное продолжение идеи явного моделирования семантических отношений предлагается в работах [307, 437], где распределенные представления используются вместе с введенными в этих работах *нейронными тензорными сетями* (neural tensor networks, NTN) для обнаружения новых семантических отношений в базах знаний. В этой модели один из уровней умеет перемножать входные векторы, что сразу резко расширяет возможности модели в плане моделирования семантических отношений. В результате NTN-модели могут обучать логические отношения (следствие, эквивалентность, отрицание и т. п.) между понятиями на основе их распределенных представлений [58, 59] (см. также применение NTN к машинному переводу в [509]).

И наконец, в-третьих. У обученных нами моделей есть один серьезный недостаток: они присваивают только один вектор каждому слову (токену). Это значит, что если у слова или биграммы окажется сразу несколько разных смыслов, что очень часто бывает в жизни, мы скорее всего обучим только один из них, а остальные потеряются. Например, в табл. 7.4 видно, что словосочетание *машина_времени* для обученной на русскоязычной «Википедии» модели word2vec означает исключительно рок-группу. Ни классический рассказ Герберта Уэллса, ни фильм «Назад

в будущее», ни вся традиция научной фантастики в целом к этому вектору отношения, по всей видимости, не имеют. Но мы не хотели бы терять эти смыслы совсем! Так возникает задача *снятия омонимии* (word sense disambiguation): как сопоставить одному слову сразу несколько векторов, а суметь различить, какой из них нужно подставить в данном контексте?

Подробный анализ таких подходов, пожалуй, выходит за рамки этой главы; отметим работу [61], в которой значения слова представляются скрытыми переменными, число значений получается из априорного распределения, заданного процессом Дирихле, а вывод ведется с помощью стохастического вариационного вывода, который мы еще обсудим в главе 10.

Еще один подход к реализации композиционной семантики, несколько ортогональный всему тому, о чем мы говорили выше, состоит в том, чтобы представлять одни слова как *модификаторы* (modifiers) других слов. Это значит, что некоторые слова представляются векторами, как и раньше, а некоторые — операциями над векторами, которые скорее изменяют значения других слов, чем имеют свои собственные значения.

Например, модели из работы [32] пытаются представить словосочетания вида «существительное + прилагательное», моделируя прилагательные как матрицы, производящие линейные операции над существительными, которые они модифицируют. Это вполне логично: например, слово «красный» представляется не вектором, а матрицей, которая делает одно и то же преобразование и над словом «дом», и над словом «мяч», сообщая им свойство «быть красными» и моделируя одним вектором словосочетания «красный дом» и «красный мяч».

Наконец, стоит упомянуть, что сама идея распределенных представлений, конечно, не ограничивается обработкой естественного языка, а пригождается и в других задачах. Например, в [373] векторы распределенных представлений применяются как базовая модель для представления поведения пользователей в Интернете: их действия соответствуют словам, а профили состоят из таких действий, как предложения и абзацы состоят из слов. Нам представляется, что можно найти и другие интересные применения для этой базовой идеи.

Итак, давайте подведем краткий итог первой части главы, посвященной распределенным представлениям слов. Последние продвижения в области распределенных представлений слов превратили этот подход в фактически прием по умолчанию в современной интеллектуальной обработке естественного языка [182]. Такие представления, на каких бы принципах они ни были основаны, по сути отображают слово из словаря в некоторый вектор в евклидовом пространстве $\mathbb{R}^d$, и основная задача распределенных представлений — попытаться выразить семантические отношения между словами в виде геометрических отношений в этом самом евклидовом пространстве.

В базовых подходах на вход модели обучения представлений слов подается корпус текстов, а на выходе обучаются векторы для каждого слова; есть и разнообразные расширения и продолжения этой идеи, но основной смысл именно таков.

Идея таких представлений сначала была применена к обычным языковым моделям, например в работах [147, 369, 442], а начиная со статей Томаша Миколова с соавторами о word2vec [123, 137] идея распределенных представлений слов начала применяться просто буквально везде, ко всем задачам современной обработки текстов.

Оказалось, что распределенные представления очень сильно помогают в большинстве классических задач обработки текстов, потому что они фактически дополняют все последующие модели, задачи и датасеты неким «сакральным знанием» о том, как слова связаны друг с другом просто в самом языке, о котором идет речь. Да и моделям обычно проще управляться с векторами из нескольких сотен вещественных чисел, чем с дискретными объектами из словаря размером в десятки или сотни тысяч. А вот насколько этого достаточно и что потом с этими векторами делать — об этом мы и будем говорить дальше.

7.5. Вверх и вниз от представлений слов

> А так как эта перекладина может быть передвигаема по желанию, вверх и вниз, на двух зубчатых рейках и закрепляется на любой высоте специальными защелками, то вполне понятно, что чем мы ниже опустим перекладину, тем более выгибается прут и тем энергичнее наносится удар. Этим мы регулируем силу наказания.
>
> *А. И. Куприн. Механическое правосудие*

В предыдущем разделе мы уже упоминали предположение о том, что вектор, равный сумме или среднему других, отражает их совместный смысл. В лингвистике этот феномен — то, что комбинации векторов отдельных слов могут отражать общий смысл больших кусков текста, — известен как *распределенная композиционная семантика* (distributional compositional semantics); у нее есть даже обоснования в когнитивной науке и некая экспериментальная база, приходящая из экспериментальной психологии [132, 367, 486].

Кроме того, в почти любой реальной задаче обработки текстов векторы слов нас интересуют постольку-поскольку, а конечная цель обычно не в том, чтобы понять что-то об отдельных словах. Например, цель анализа тональности (сентимент-анализа) состоит не в том, чтобы выяснить, что «отлично» — это позитивное слово, а «так_себе» — негативная биграмма, хотя это может быть важным промежуточным шагом, а в том, чтобы понять тональность целых текстов, пусть коротких.

Поэтому естественным следующим действием после обучения представлений отдельных слов будет попытка выразить смысл более крупных участков текста: предложений, абзацев, статусов в социальных сетях, статей и т. д. Для этого нужно найти способ объединить векторы отдельных слов в векторы нескольких слов и т. д.; этому посвящен целый ряд разных работ, и в этом разделе мы начнем с того, что дадим краткий обзор некоторых из таких моделей.

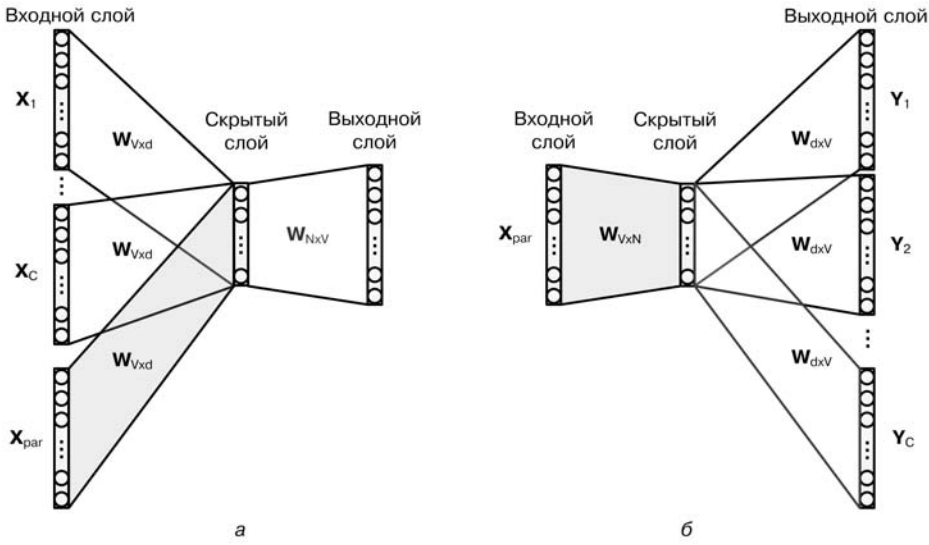


Рис. 7.3. Архитектуры для векторов участков текста из работы [301]: *а* — PV-DM; *б* — PV-DBOW

Самая простая идея, первой приходящая в голову, — это использовать простые комбинации векторов слов, сумму или среднее, для того, чтобы представить предложение или абзац в виде вектора. Такой подход кажется чрезвычайно наивным, но не стоит забывать, что одной из основных целей обучения векторов word2vec и GloVe было обучить такие представления, в которых семантические соотношения между понятиями будут выражаться именно простыми геометрическими соотношениями между векторами. Поэтому хотя такой подход обычно используется как baseline, то есть подлежащий улучшению стандартный наивный подход (см., например, [301]), в [123] он отмечается как разумный подход для получения представлений коротких фраз, а в более поздней работе [149] было показано, что он весьма эффективен для резюмирования документов (summarization). Таким образом, если вы взялись реализовывать современную систему обработки текстов, и вам нужно представить предложение, абзац, твит или другой короткий кусочек текста в виде вектора, в первую очередь попробуйте просто усреднить векторы входящих в него слов: не исключено, что в результате вы получите вполне рабочий вариант модели, и ничего другого делать будет просто не нужно.

Второй класс моделей, обучающих векторы представлений текстов, выглядит как обучение векторов слов, но с дополнительными параметрами, соответствующими более крупным кускам текста. Такие модели были подробно разобраны в работе [301], где вводится два варианта обучения так называемых *paragraph vectors* (векторов абзацев) в процессе обучения word2vec (рис. 7.3). В модели PV-DM (Distributed Memory Model of Paragraph Vectors) вектор, представляющий абзац

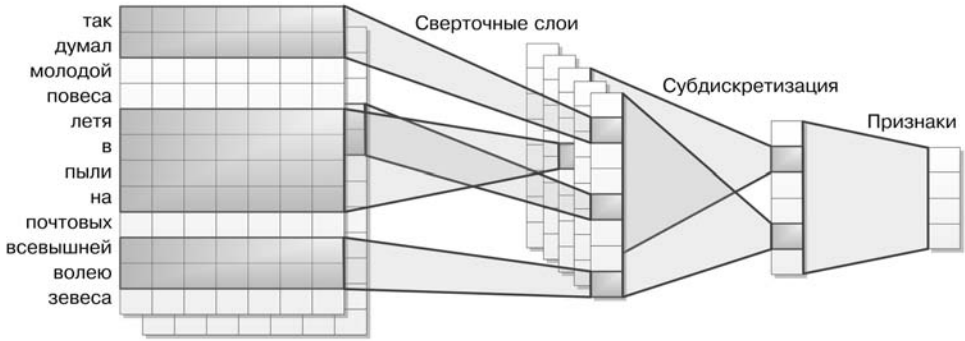


Рис. 7.4. Одномерная сверточная сеть для обработки текстов

или предложение, определяется как еще один дополнительный вектор весов. Он работает в качестве «памяти», сохраняя более долгосрочный контекст для векторов слов, чем можно найти в маленьком окне; схема показана на рис. 7.3, *а* (мы закрасили часть, относящуюся к вектору участка текста). А в модели PV-DBOW (Distributed Bag of Words Model of Paragraph Vectors) [301] слова контекста просто полностью игнорируются, и модель использует вектор абзаца, чтобы предсказать слова из окна в том же абзаце; PV-DBOW показана на рис. 7.3, *б*.

Третья интересная мысль состоит в том, чтобы «поднять» идею предсказания слова по локальному контексту на уровень выше. Например, в работе [498] предлагаются так называемые *skip-thought* векторы: смысл предложения превращается в вектор с помощью skip-gram конструкции, построенной на целых предложениях с помощью заранее обученных представлений отдельных слов. А в [213] распределенные представления продолжаются на уровень целых документов; эта работа посвящена обработке больших потоков коротких текстов (например, *Twitter*), и в ней строится иерархическая языковая модель с отдельными уровнями документа и токена (слова).

Наконец, конечно, почти любой пример реального применения нейронных сетей в обработке естественного языка будет в какой-то момент представлять короткие участки текста в виде векторов фиксированной размерности. В разделе 8.1 мы поговорим о так называемых *encoder-decoder* архитектурах, в которых текст (обычно короткий) сначала «сворачивается» до одного вектора, а затем «разворачивается» обратно в виде результата: того же текста на другом языке, ответной реплики в диалоге... Это, конечно, более конкретный класс моделей, но практически любое применение к определенной практической задаче так или иначе содержит тот самый encoder, кодировщик.

Отметим здесь, что хотя более «естественной» архитектурой для обработки текстов представляются рекуррентные сети (в конце концов, текст — это такая последовательность), в последние годы сверточные сети также активно применяются

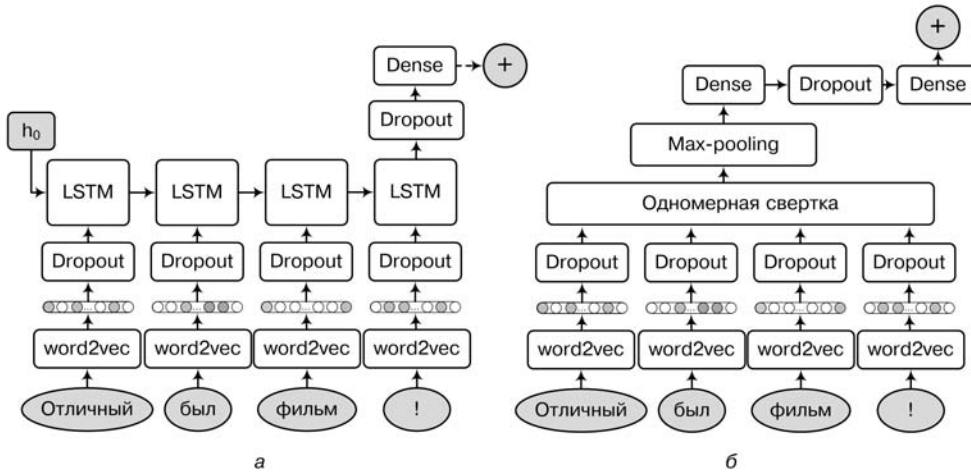


Рис. 7.5. Примеры нейронных сетей для задачи анализа тональности [64]: *а* — рекуррентная; *б* — сверточная

для этой цели и считаются ничем не хуже, а может быть, и лучше. Основная разница между сверточными сетями для обработки текстов и теми сетями, которые мы подробно рассматривали в главе 5, состоит в том, что свертки теперь одномерные, то есть окно считается «в длину», по словам предложения. На рис. 7.4 изображен пример первого сверточного слоя для обработки текста: слова сначала превращаются в векторы теми или иными вложениями, а затем к этим векторам чисел применяются одномерные свертки.

А в качестве конкретного примера мы изобразили на рис. 7.5, *а* простейшую нейронную архитектуру для задачи анализа тональности; она настолько логична и прямолинейна, что приводится буквально в документации библиотеки Keras как пример задания рекуррентных сетей [79], а код ее очень краток и понятен. Давайте приведем его; в этом разделе мы адаптируем пример, приведенный в [64]. Начнем с того, что импортируем все, что нужно, из Keras и загрузим стандартный датасет отзывов на фильмы из базы *IMDB* для анализа тональности. Мы ограничимся словарем из 5000 самых часто встречающихся в этом корпусе слов, а также обрежем все отзывы до 500 слов. Мы также дополним более короткие отзывы нулевыми векторами до длины 500; для этого в Keras есть удобная функция `pad_sequences`.

```
import numpy
from keras.datasets import imdb
from keras.models import Sequential
from keras.layers import Dense,LSTM
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
top_words = 5000
```

```
max_review_length = 500
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=top_words)
X_train = sequence.pad_sequences(X_train, maxlen=max_review_length)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
```

Теперь построим сеть. Мы не будем загружать готовые представления слов, а обучим их отдельно как первый уровень модели. Поскольку датасет маленький, представления размерности 200–300 обучить не получится, да и незачем, мы ограничимся векторами длины 32. Вслед за ними зададим архитектуру, изображенную на рис. 7.5, *а*; в Keras это очень просто.

```
embedding_vector_length = 32
model = Sequential()
model.add(Embedding(top_words, embedding_vector_length,
                    input_length=max_review_length))
model.add(Dropout(0.2))
model.add(LSTM(100))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))
```

Теперь можно обучить и посмотреть, что получается на тестовом множестве.

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=3, batch_size=64)
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.4f" % (scores[1]))
```

Точность предсказания тональности у нас получилась 85,42 % (естественно, точное число подвержено случайным факторам).

Обратите внимание, что при этом окончательное представление отзыва опять выглядит как вектор фиксированной длины, просто теперь в него «сворачивается» весь отзыв, сначала проходя через слой вложений (распределенных представлений), а затем через рекуррентный слой. Конечно, эту архитектуру можно усложнять и дальше; скорее всего, получится что-то похожее на рис. 6.13, который мы подробно обсуждали в главе 6.

А можно существенно ускорить обучение, выиграв при этом и в качестве, с помощью сверточной архитектуры; свертки в ней будут одномерные, но мы точно так же будем начинать с векторов слов и двигаться вверх к представлению всего отзыва. Эта архитектура изображена на рис. 7.5, *б*; а в коде изменится совсем немного:

```
model = Sequential()
model.add(Embedding(top_words, embedding_vecor_length,
                    input_length=max_review_length))
model.add(Conv1D(filters=32, kernel_size=3, padding='same', activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
```

Все остальное выглядит точно так же, но сеть теперь обучается гораздо быстрее (сверточные сети обучать проще и быстрее, чем рекуррентные, а тут мы для рекуррентной части сократили входную размерность), а итоговая точность на тестовом множестве в наших экспериментах составила 87,35 %.

Итак, мы увидели примеры того, как строить классические нейросетевые архитектуры на основе распределенных представлений слов. Такие архитектуры часто и успешно применяются (см., например, [271]). Однако в этой части книги нас больше интересуют ситуации, когда задачи обработки естественных языков привели к появлению новых, нестандартных архитектур нейронных сетей. К ним мы перейдем буквально в следующем разделе, но сначала — немного критики.

Распределенные представления слов, которые мы до сих пор обсуждали, имеют и ряд важных недостатков. Во-первых, векторы для каждого слова совершенно независимы: мы не можем использовать наши знания об одном слове, чтобы лучше понять другое. Это не кажется страшным, пока вы не задумываетесь о языках с богатой морфологией вроде русского: для модели *word2vec* слова «вектор», «вектора», «векторы», «векторный», «векторного», «векторизовать», «подвектор» и т. п. являются абсолютно разными (это просто разные размерности на входе), и для *каждого* из них придется набирать достаточно статистики, подбирать такой датасет, чтобы каждая форма встречалась несколько раз. Однако мы с вами прекрасно понимаем целое «гнездо» слов, как только узнаем, что означает его базовый корень; можно ли это умение как-то передать компьютеру?

Во-вторых, то же самое относится к словам, которых нет в словаре: невозможно обучить распределенное представление слова без достаточного набора данных про него, в то время как человек легко может понять, что в эитх солвах бли допущны оппечатки, а зачастую может и экстраполировать смысл слова из его формы. В качестве примера мы придумали наукообразно звучащее слово *polydistributional*: когда один из авторов книги начал использовать это слово в докладах, Google давал всего 48 ссылок с этим словом даже без кавычек — как вы понимаете, это значит, что слово не было известно ровным счетом никому. Однако вы ведь уже примерно понимаете, что оно значит? Мы тоже. И действительно, в единственной научной статье из этих 48 результатов слово *polydistributional* не вводилось как новый термин, а просто использовалось между делом в тексте, в предположении, что оно всем понятно... и оно действительно всем понятно[1].

В-третьих, на практике модели распределенных представлений слов становятся очень объемными для больших словарей. Хотя применять обученную модель даже очень большого размера можно достаточно быстро (применение сводится к тому, чтобы достать из памяти нужный вектор по ключу), чтобы это осуществить в реальности, придется все векторы загрузить в память, что не всегда так легко.

---

[1] В этой книге мы тоже провели такой эксперимент: говоря о Дартмутском семинаре в разделе 1.2, мы назвали заявление с просьбой поддержать его проведение «грантозаявкой». На это слово сейчас (лето 2017 года) Google выдает меньше десяти ссылок, и все на наши собственные презентации... но неужели вы не поняли, о чем речь?

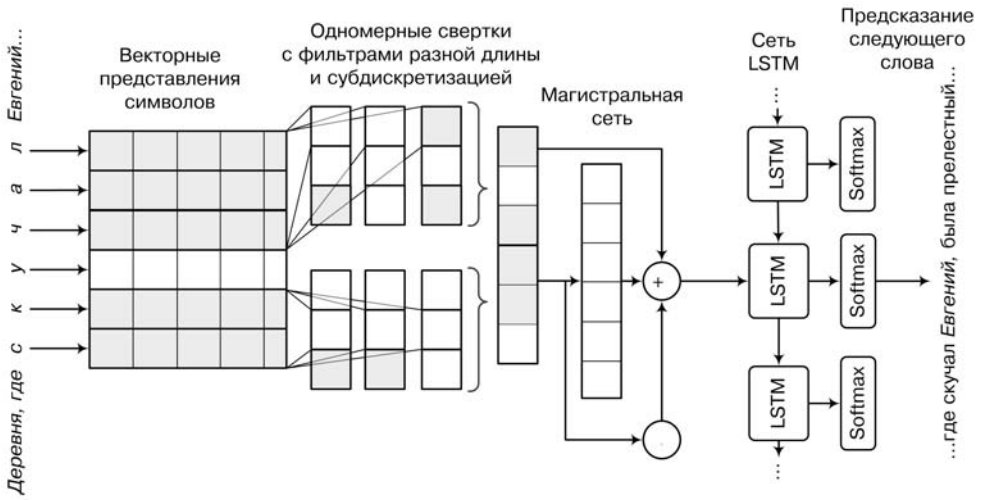


Рис. 7.6. Структура посимвольной языковой модели из работы [74]

Все эти проблемы приводят к идее *посимвольных представлений (character-level representations)*: от векторов слов можно идти не только «вверх», в сторону векторов предложений и абзацев, но и «вниз», спускаясь с уровня слов на уровень отдельных символов. Давайте постараемся обучить модель, которая может понимать структуру слова и делать вывод о том, что «вектор», «вектора» и «векторы» должны находиться не так далеко друг от друга в семантическом пространстве.

Первые попытки учесть структуру слова состояли в том, чтобы попытаться разложить слово на *морфемы*, наименьшие смысловые части письменного языка (как в шестом классе: приставка, корень, суффикс, окончание...) [54, 340, 501]. Если бы морфемы были доступны напрямую, они действительно стали бы идеальными «кирпичиками» для низкоуровневых представлений слов. Однако на самом деле морфемы непосредственно в ощущениях нам не даны; морфологический разбор — это отдельная непростая задача, и в каком-то смысле мы просто смещаем ее сложность на морфологический анализатор, который тоже не всегда будет работать идеально. Кроме того, важной мотивацией для посимвольных представлений является также обработка опечаток и ошибок, которые могут оказаться в любой морфеме, в том числе и в корне.

Поэтому современные посимвольные модели работают непосредственно с буквами. В работе [156] представлена базовая модель C2W (character to word), которая может обучать представления слов из составляющих их букв с помощью двунаправленных LSTM. В этой модели поданные на вход символы сначала преобразуются в соответствующие им векторы (да, тут тоже есть такая же матрица распределенных представлений, но не для слов, а для отдельных букв), а затем векторы подаются в двунаправленный LSTM, который уже выдает представление слова. Все

это можно теперь обучать одновременно (end-to-end), все градиенты отлично проходят, и все работает: в [156] показаны отличные результаты для задач языкового моделирования и частеречной разметки, особенно для языков с богатой морфологией.

А другой естественный способ работать с последовательностями — это одномерные сверточные сети. Например, в модели Text understanding from scratch («Понимание текста с нуля» — название, пожалуй, слишком громкое) [586, 587] отдельные символы в векторном представлении подаются в сверточную сеть с шестью сверточными слоями, тремя полносвязными слоями и двумя слоями дропаута для регуляризации. Утверждается, что такая модель также показывает очень хорошие результаты в классификации текстов и других классических задачах обработки текстов.

Современные посимвольные модели обычно сочетают все эти архитектуры, а также добавляют новые трюки. На рис. 7.6 мы показали характерный пример современной посимвольной модели [74], в которой строится посимвольная нейросетевая языковая модель. Обратите внимание, как много в этой архитектуре сходится из того, о чем мы говорили раньше: сначала распределенные представления отдельных букв, затем одномерные свертки с субдискретизацией по времени, затем конкатенированные признаки из сверточной части подаются в так называемую *highway network* [507], основная компонента которой — остаточные связи наподобие Resnet, а затем полученные представления слов подаются в рекуррентную сеть на основе LSTM, которая предсказывает следующее слово. И снова мы видим магию нейронных сетей: весь этот зоопарк разных моделей совершенно без проблем уживается вместе, и вся модель может обучаться одновременно, потому что вся эта сложная конструкция — всего лишь большая композиция простых функций, и методом обратного распространения градиент функции ошибки подсчитать достаточно легко.

Заметим, что применять посимвольную модель обычно вычислительно сложно: ни одно реальное применение обработки естественного языка не будет работать, если потребуется запускать двунаправленный LSTM для каждого слова отдельно. К счастью, это не проблема: никто не запрещает нам предвычислить представления любого требующегося числа слов, а потом в реальном применении запускать модель только для редких слов, которые в этот словарь не попали. Тем самым посимвольные модели дают очень естественный способ варьировать баланс между размером словаря (то есть фактически требующейся памятью) и быстродействием модели.

Заметим, что модели, основанные на отдельных буквах, неизбежно должны использовать в качестве входа *последовательность* букв и работать с ней как с последовательностью, то есть либо рекуррентными, либо сверточными сетями. Вопреки популярному наблюдению, мы не можем просто взять и забыть порядок букв в слове и представлять слова векторами числа букв размерности несколько десятков: будет слишком много коллизий и мало информации.

Однако оказывается, что можно найти золотую середину. Промежуточный подход был развит в работах от *Microsoft Research*, посвященных так называемым *глубоким структурированным семантическим моделям* (Deep Structured Semantic Models, DSSM) [297, 372]. В качестве нижнего уровня обработки DSSM используют вложения частей слов (subword embeddings), которые представляют слово как набор буквенных триграмм; например, «слово» кодируется как {#сл, сло, лов, ово, во#}. При таком подходе словарь сокращается до $|A|^3$, где A — алфавит. В русском языке, например, $33^3 = 35\,937$, а если учесть, что далеко не все триграммы реально встречаются, и того меньше. Основной же бонус от такого подхода состоит в том, что теперь можно забыть порядок триграмм, и слова все равно не перепутаются: в [304] рассматривается корпус текстов с 500 тысячами слов (такие гигантские словари часто набираются за счет опечаток), и авторы насчитали ровно 22 коллизии на все 500 тысяч, то есть 22 случая, когда разные слова имели одинаковое множество буквенных триграмм. А размерность сократилась с 500,000 до 30. Очевидно, что такое представление достаточно устойчиво и к небольшим ошибкам и опечаткам: изменение одной буквы заденет только три триграммы; это крайне важно, например, при анализе любых текстов, порожденных интернет-пользователями.

Интересным направлением для дальнейших исследований остается то, как наилучшим образом сочетать распределенные представления слов и модели, основанные на символах (см., например, [262]). Отметим работу [146], в которой LSTM на векторах слов дополнялся LSTM на символьных представлениях, и в результате получались отличные результаты в языковом моделировании; а в [387] аналогичный подход применялся к распознаванию именованных сущностей.

Тем не менее, самым, наверное, интересным развитием идей вложения частей слов стала работа с участием все того же Томаша Миколова [144], которая стала основой для реализации распределенных представлений слов в очень популярной библиотеке *FastText* [21, 153]. Идея чрезвычайно проста: давайте использовать все те же идеи word2vec-моделей, но в качестве базовых векторных представлений будем искать представления не слов, а триграмм символов, как в DSSM. Это значит, что мы вводим векторные представления для триграмм символов, затем раскладывает каждое слово как сумму векторов его триграмм, а на этих суммах строим обычные конструкции CBOW или skip-gram. Полученные модели обучить проще и быстрее, они не такие большие (мы уже говорили, что триграмм символов куда меньше, чем разных слов), и учет морфологии в них тоже получается почти автоматически. Оказывается, что в результате получаются представления, работающие заметно лучше для языков с богатой морфологией. Пожалуй, именно эти представления мы рекомендуем в настоящее время использовать для русского языка (только их придется обучить самостоятельно, так что готовьте большой корпус).

Итак, представления слов и посимвольные модели — это современное состояние обработки естественных языков. Сейчас это важная и еще далеко не исследованная до конца область, в которой все время появляются новые результаты,

и мы уверены, что в скором будущем появятся новые варианты представлений слов и других языковых объектов, а также еще более убедительные модели, умеющие читать слова по частям. Но интересные нейросетевые архитектуры в обработке текстов на этом только начинаются...

7.6. Рекурсивные нейронные сети и синтаксический разбор

> Мы выдвинули впервые новые принципы творчества, кои нам ясны в следующем порядке:
>
> 1. Мы перестали рассматривать словопостроение и словопроизношение по грамматическим правилам, став видеть в буквах лишь *направляющие речи*. Мы расшатали синтаксис.
>
> *Д. Бурлюк и др. Манифест из альманаха «Садок судей»*

До сих пор все архитектуры для обработки текста, которые мы встречали в этой книге, начинались с того, что рассматривали текст как *последовательность*: или слов, или сразу символов. Собственно, и рекуррентные архитектуры так хорошо подходят для обработки текстов именно потому, что предназначены для обработки последовательностей: например, сеть LSTM-ячеек последовательно читает слово за словом (точнее, вектор за вектором) и постепенно кодирует все предложение в один скрытый вектор, который затем используется для, например, анализа тональности.

Но если мы внимательнее присмотримся к тому, как устроен текст, мы увидим, что на самом деле его структура вовсе не представляет собой последовательность! Мы уже видели это в начале главы, когда говорили о грамматиках Хомского: предложения на естественном языке более естественно представляются в виде *дерева*. Вспомните, как в школе вы разбирали предложение: подлежащее, сказуемое, дополнение... Разные слова относятся друг к другу, а последовательность, в которой они идут в предложении, с этим связана только весьма косвенно. Например, в предложении «мама мыла раму с мылом» слова «с мылом» зависят от «мыла», а вовсе не от «рамы», и это словосочетание тоже можно разобрать дальше. Получается не последовательность, а *дерево*, примерно как на рис. 7.1, *б*. Такие деревья являются результатом синтаксического анализа на основе *грамматики зависимостей* (dependency parsing), и в наше время, когда речь идет о синтаксическом анализе, обычно имеют в виду построение именно таких деревьев.

Может быть, это значит, что мы можем использовать эту структуру дерева, и нам будет легче понять то, что говорится в предложении? Такая проблема кажется особенно важной для анализа тональности: для того чтобы понять, позитивно настроен автор текста или негативно, может быть очень важно разобраться, к какому именно слову относится брошенное им «не».

Такой подход действительно достаточно активно разрабатывается. В качестве упрощенного примера выделим работу [118], где сверточные сети моделируют последовательности с помощью n-грамм, учитывающих структуру синтаксических зависимостей. Но самое важное развитие этой идеи — так называемые *рекурсивные нейронные сети* (recursive neural networks) [134, 483]; не путать с рекуррентными! Это один из ярких примеров того, как практическая задача обработки естественного языка привела к совершенно новой архитектуре сети.

Основная идея рекурсивной сети состоит в том, чтобы шаг за шагом идти по дереву снизу вверх и постепенно «собирать» дерево к своему корню: на каждом уровне сеть берет представления очередных узлов и объединяет их. В базовом варианте мы предполагаем, что на вход рекурсивная сеть получает бинарное дерево синтаксического разбора предложения, а затем представления на каждом шаге объединяются.

Формально говоря, мы представляем вершину дерева v вектором $\boldsymbol{x}_v$ и строим рекурсивную сеть так:

$$\boldsymbol{x}_v = f(W_L \boldsymbol{x}_{l(v)} + W_R \boldsymbol{x}_{r(v)} + \boldsymbol{b}),$$

где $l(v)$ — левый потомок вершины v, $r(v)$ — ее правый потомок, W_L и W_R — соответствующие матрицы весов, $\boldsymbol{b}$ — вектор свободных членов, а f — нелинейная функция активации, обычно логистический сигмоид или ReLU.

Обратите внимание, что матрицы W_L и W_R остаются одними и теми же на всем дереве, мы каждый раз применяем одно и то же преобразование — отсюда и слово «рекурсивный». Таким образом, по сути рекурсивная сеть — это вариант рекуррентной сети с достаточно хитрой архитектурой соединений. И обучается она так же, как рекуррентная сеть, аналогом алгоритма обратного распространения во времени, только вместо времени теперь структура дерева, и градиенты путешествуют от корня, где вычисляется вектор всего предложения и через него подсчитывается результат сети и функция ошибки, к листьям, где расположены векторы слов. Все это мы постарались проиллюстрировать на рис. 7.7, где приведен пример анализа тональности для очень простой фразы. У каждого узла мы также разместили потенциально «правильную» оценку его тональности: обратите внимание, что хотя слово «не» само по себе не является ни позитивным, ни негативным, оно должно менять окраску второго поддерева на противоположную.

Вспомним теперь, как мы строили глубокие рекуррентные сети, в которых обычная глубина «во времени» дополнялась глубиной «в пространстве», достигаемой несколькими рекуррентными уровнями друг над другом. Точно то же самое можно сделать и для рекурсивных сетей, реализовав одновременно глубину в графе и глубину нескольких уровней. Глубокие рекурсивные сети были предложены в работе [253]. Теперь у каждой вершины v, находящейся на уровне i, три «потомка»: узлы того же уровня, соответствующие ее настоящим потомкам в дереве $l(v)$ и $r(v)$, а также узел, соответствующий той же самой вершине дерева, но на предыдущем уровне $i-1$.

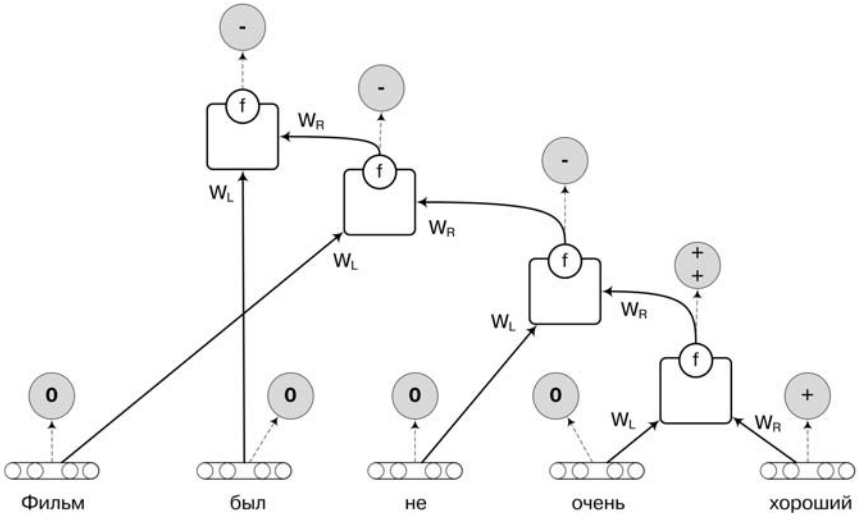


Рис. 7.7. Структура рекурсивной нейронной сети

Мы проиллюстрировали архитектуру глубоких рекурсивных сетей на рис. 7.8: структура достаточно сложная, и чтобы проще было ее проследить, мы нарисовали серым цветом все, что относится к первому уровню, а черным — ко второму; пунктирные линии показывают связи между уровнями.

Есть только одна техническая разница: в обычной рекурсивной сети мы представляли каждый узел, начиная с листьев, вектором одной и той же размерности. Эта размерность определялась листьями и в точности соответствовала размерности распределенных представлений отдельных слов.

А теперь мы разделим представление листьев $\boldsymbol{x}_v$ и внутренних узлов сети $\boldsymbol{h}_v$. Минус этого решения в том, что нам придется обучить по две разных матрицы W_L и W_R, отдельно для случая листьев и для случая композиции внутренних узлов. А плюс в том, что мы теперь можем выбрать какую захотим размерность для представления внутренних узлов дерева; обычно ее выбирают существенно меньшей, чем размерность векторов слов. В случае глубоких рекурсивных сетей оказывается, что плюс существенно перевешивает минус.

Теперь формально: «глубина» представляется в виде дополнительных матриц весов $V^{(i)}$, которые связывают вершины i-го уровня с их предшественниками на $(i-1)$-м уровне. Кроме того, матрицы W_L и W_R тоже теперь становятся разными для разных уровней; число параметров существенно увеличивается, но общая формула остается достаточно простой. На уровне i:

$$\boldsymbol{h}_v^{(i)} = f\left(W_L^{(i)}\boldsymbol{h}_{l(v)}^{(i)} + W_R^{(i)}\boldsymbol{h}_{r(v)}^{(i)} + V^{(i)}\boldsymbol{h}_v^{(i-1)} + \boldsymbol{b}^{(i)}\right).$$

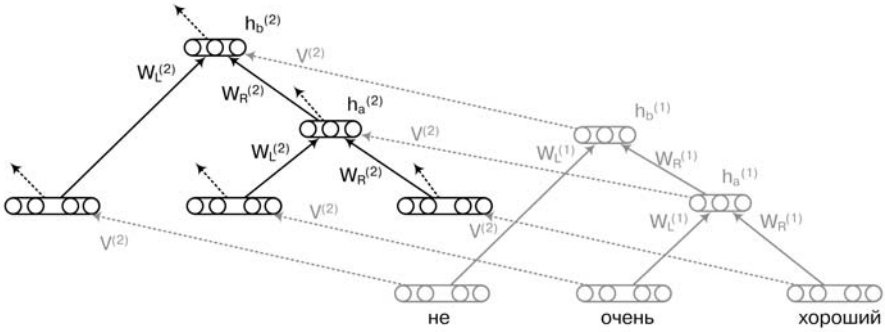


Рис. 7.8. Глубокая рекурсивная нейронная сеть

Глубокие рекурсивные сети улучшают анализ тональности еще сильнее.

Кстати, эта идея представлять слова матрицами, которые будут умножаться на вектора других слов, используется не только в рекурсивных нейронных сетях. Она также появляется и в еще одном классе подходов к обработке композициональной семантики, который можно считать промежуточным решением, не доходящим до полной и законченной рекурсивности. В ряде работ было предложено принимать некоторые слова за *модификаторы* других слов и представлять их как операции на векторах. Грубо говоря, мы считаем, к примеру, существительные «дом» или «автомобиль» векторами, а прилагательные «красный» или «большой» — операциями, которые их модифицируют, превращая семантику «дома» в семантику «большого дома» или семантику слова «автомобиль» в семантику словосочетания «красный автомобиль».

В простейшем варианте эти модификаторы, конечно, разумно считать линейными. Тогда слова-модификаторы будут представлять собой матрицы, выражающие эти линейные операторы. Такой подход применялся к словосочетаниям вида «существительное — прилагательное» в работе [32]. Есть и более сложные подходы, в которых строятся абстрактные категорные модели композициональной семантики и предлагаются процедуры для совмещения семантики слов в семантику предложений [83, 84, 87]. Например, в работе [197] слова, выражающие отношения, тоже моделируются матрицами, но дальше предлагается специальная процедура для того, чтобы объединять эти представления в векторы предложений; похожие результаты были получены и в [62, 92, 275, 462].

Мы не будем подробно рассматривать эти подходы, отметим здесь только, что в последние годы это направление исследований обогащается работами, в которых глаголы представляются тензорами малого ранга [165], мультиязыковыми моделями [211], моделированием семантической композиции смыслов через композицию функций [409] и другими подходами к формальной распределенной семантике [93, 196, 367].

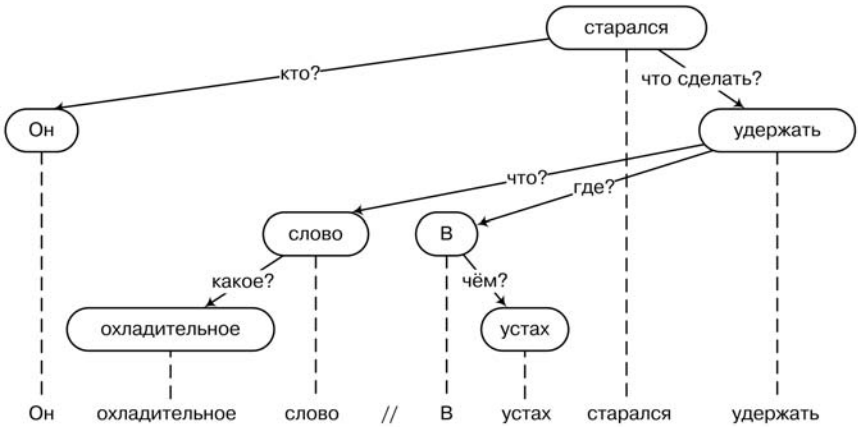


Рис. 7.9. «Он охладительное слово // В устах старался удержать»: непроективное дерево разбора

Но закончить разговор о рекурсивных сетях придется на минорной ноте. Как видите, хотя они выглядят очень естественным способом обработки естественного языка — дерево разбора предложения действительно будет гораздо более приближенным к семантике представлением, чем просто последовательность слов, — для работы этим моделям на вход нужно подавать то самое дерево... а откуда оно возьмется? Естественно, существуют модели, которые делают синтаксический разбор автоматически, и мы к ним перейдем чуть ниже. Но они тоже могут ошибаться. В любом случае, получается, что для того чтобы хорошо решить одну задачу (скажем, анализ тональности), нам еще нужно по дороге научиться неплохо решать другую (синтаксический разбор), ничуть не менее сложную.

Чтобы проиллюстрировать сложности, возникающие при синтаксическом разборе, давайте в качестве небольшого лирического отступления посмотрим на два примера из нашего всего, знаменитого «Евгения Онегина».

Первый пример, на рис. 7.9, не кажется сложным, однако он представляет почти непреодолимые трудности для многих современных парсеров. Дело в том, что большинство алгоритмов синтаксического разбора могут работать лишь с соседними токенами, то есть строить только такие деревья, в которых ребра не пересекаются с проекциями слов (на рисунке проекции показаны пунктирными линиями); такие деревья называются *проективными*. А здесь проективное дерево никак не получится: «охладительное слово» и «в устах» относятся к глаголу «удержать», который зависит от «старался», и тут без пересечений не обойтись.

Но и это еще полбеды. Главная проблема состоит в том, что в живом языке возникает немало ситуаций, когда разбор неоднозначен. Могут быть ситуации чистой неоднозначности, вроде «по деревне шла девушка с косой», когда даже человек ничего не может поделать, так что и алгоритму мы это в вину не поставим. Но чаще

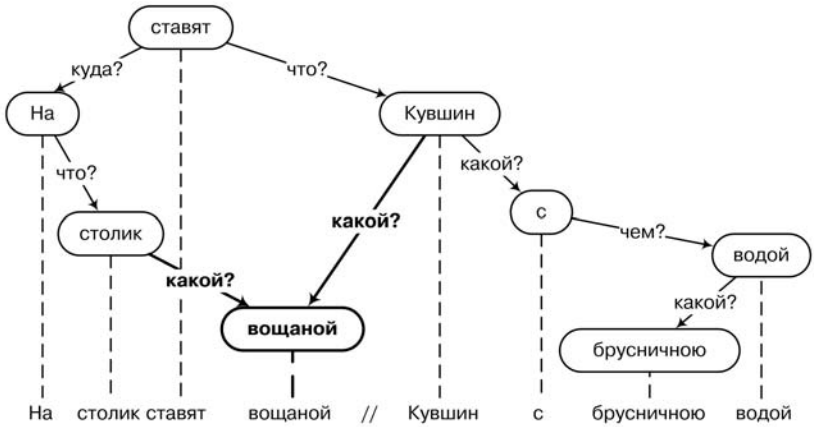


Рис. 7.10. «На столик ставят вощаной // Кувшин с брусничною водой»: неоднозначный разбор

бывает так, что человеку все понятно благодаря тому, что он понимает *смысл* слов и текста в целом... а с этим у компьютерных программ, как мы уже обсуждали, пока что очень плохо. На рис. 7.10 — характерный пример из «Евгения Онегина»: кто здесь вощаной, столик или кувшин? По синтаксису здесь, кстати, больше похоже на кувшин, и если бы вместо «вощаной» стояло «стеклянный», мы бы вряд ли подумали о стеклянном столике, даже в современном тексте. Но мы с вами знаем, что «вощаным» кувшин не бывает, а столик бывает, и это позволяет нам проанализировать предложение правильно и однозначно... а бедному компьютеру что делать?

Можно цинично предположить, что успехи рекурсивных нейронных сетей для анализа тональности не в последнюю очередь обусловлены наличием отличного датасета по анализу тональности — Stanford TreeBank [443]. В этом датасете уже даны деревья синтаксического разбора для входящих в него предложений; более того, метки тональности проставлены даже для промежуточных узлов деревьев разбора, а не только для всех предложений целиком! Естественно, на таком датасете рекурсивные сети работают прекрасно, однако для других задач картина не столь радужная. Так, практика показывает, что для вышеупомянутого распознавания именованных сущностей рекуррентные и сверточные сети обычно работают лучше рекурсивных. Более того, сверточную сеть в данном случае можно рассматривать как своеобразный «мягкий» парсер: хоть она и не может переставлять слова в нужном порядке, но все-таки уровень за уровнем выделяет все более общие признаки из исходного текста. Поэтому в последнее время о рекурсивных моделях для обработки текстов слышно не так много. Однако мы решили все равно кратко упомянуть их в книге, потому что нам представляется, что аналогичные подходы могут оказаться очень полезны для обработки других видов данных, которые можно естественным образом представить в виде дерева.

Итак, мы увидели, что синтаксический разбор — это ключевая часть предобработки для многих задач обработки естественного языка. Но, может быть, глубокие нейронные сети могут помочь и с самим разбором тоже? Здесь возникает еще одна любопытная новая архитектура нейронных сетей, которую мы опишем очень кратко. Большинство современных синтаксических парсеров имеют непрерывное состояние (continuous-state parsers), то есть кодируют текущее состояние алгоритма разбора как вектор в евклидовом пространстве [27, 67, 391, 510, 512, 535, 539, 540]. А сами алгоритмы основаны на стеке, в который помещаются поддеревья разбора, накопившиеся к текущему моменту; суть алгоритма — выбрать, в каком порядке помещать части предложения в стек и доставать их из него. Поэтому в работе [540] строится модель *Stack LSTM*, которая моделирует стек, выстраивая его из LSTM-ячеек, и отдельные нейронные сети управляют тремя структурами данных:

- буфер B состоит из последовательности слов и представлен как вектор $\boldsymbol{b}_t$ во время t;
- стек S хранит частичные деревья разбора; это вектор $\boldsymbol{s}_t$ во время t;
- список A состоит из действий, которые уже предпринял парсер; это вектор $\boldsymbol{a}_t$ во время t.

А векторы $\boldsymbol{b}_t$, $\boldsymbol{s}_t$ и $\boldsymbol{a}_t$, в свою очередь, представлены как скрытые состояния соответствующих Stack LSTM. Особенность Stack LSTM состоит в том, что к обычной цепочке LSTM-ячеек добавляется «указатель на стек», который указывает на то, какой именно выход будет прочитан; см. рис. 7.11, на котором x_i обозначает содержимое стека. Операция pop просто передвигает указатель налево, а при операции push новая LSTM-ячейка добавляется справа от текущей позиции указателя. Соответственно, чтобы проивести операцию reduce, то есть объединить два дерева, модель использует рекурсивную сеть, соединяя векторы двух поддеревьев из S в новый вектор, который затем помещается в S.

Важное расширение Stack LSTM состоит в том, чтобы учесть морфологию. Морфология (когда она есть), очевидно, очень важна для синтаксического разбора. Базовая модель в [540] выдавала отличные результаты для английского языка, представляя каждое слово его вектором $\boldsymbol{w}$ и тегом части речи $\boldsymbol{t}$, который подавался на вход модели отдельно; неизвестные слова представлялись отдельным токеном UNK.

Следующая работа об этом [27] рассматривала синтаксический разбор в языках с богатой морфологией, и там базовые представления слов производились двунаправленными LSTM на уровне отдельных символов, совсем как в [156]. Полученные представления действительно хорошо отражают морфологию и заметно улучшают синтаксический разбор; любопытно, правда, что в длинном списке морфологически богатых языков в [27] русского не нашлось — видимо, этот досадный пробел придется заполнять вам, читатели.

Это интересная архитектура, но, конечно, не единственная. Многие современные подходы к синтаксическому разбору и анализу зависимостей используют ненаправленные графические модели (conditional random fields, CRF) [289, 518].

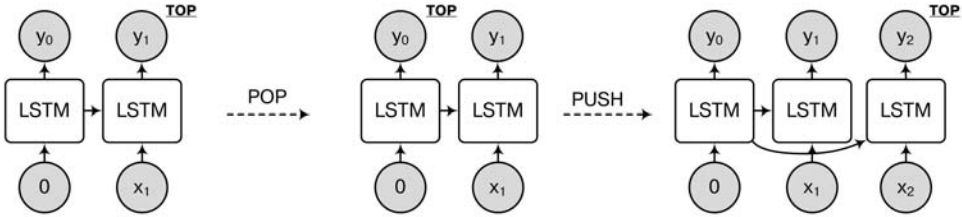


Рис. 7.11. Stack LSTM: pop, а затем push [540]

Современные парсеры, основанные на CRF, иногда работают даже лучше, чем нейронные модели [202]. В работе [133] CRF сочетаются с нейронными сетями: потенциалы на ребрах моделируются нейронной сетью, а не линейными функциями от признаков, как раньше. В любом случае, очевидно, что хороший синтаксический разбор заметно улучшает результаты обработки естественного языка.

Итак, в этой главе мы поговорили об основах современной обработки естественных языков: распределенных представлениях слов и синтаксическом разборе. Мы выбрали эти темы из множества разных задач потому, что они представляют собой основу всего того, что происходит дальше: на обычных или посимвольных представлениях слов основаны практически все современные архитектуры для работы с естественными языками. И некоторые из этих архитектур мы встретим уже в следующей главе.

Глава 8
Современные архитектуры,
или Как в споре рождается истина

TL;DR

Эта глава посвящена архитектурам глубоких нейронных сетей, появившихся в последнее десятилетие. Мы:

- поговорим о сетях с вниманием и архитектурах с кодировщиком и декодировщиком;
- обсудим разные типы порождающих моделей и поговорим о том, как глубокие нейронные сети могут помочь с порождением объектов, а не только распознаванием;
- основную часть главы посвятим порождающим состязательным сетям (GAN), в которых генератор пытается обмануть дискриминатор с помощью своих сгенерированных примеров;
- в частности, рассмотрим теорию и практику GAN и увидим пример состязательного автокодировщика (AAE).

8.1. Модели с вниманием и encoder-decoder

Белогубов. Мне, Аким Акимыч, только бы обратили внимание.
Юсов (строго). Что ты шутишь этим, что ли?
Белогубов. Как можно-с!..
Юсов. Обратили внимание... Легко сказать! Чего еще нужно чиновнику? Чего он еще желать может?
Белогубов. Да-с!
Юсов. Обратили на тебя внимание, ну, ты и человек, дышишь; а не обратили — что ты?
Белогубов. Ну, что уж-с.
Юсов. Червь!

А. Н. Островский. Доходное место

В этой главе мы начинаем рассматривать основные архитектуры, которые появились в нейронных сетях в последние годы. Если раньше мы в основном объясняли, как заставить хорошо работать классические архитектуры, созданные в 1980-х годах, то сейчас начнем двигаться дальше базовых конструкций полносвязных, сверточных и рекуррентных нейронных сетей и посмотрим, что можно к ним добавить так, чтобы результаты стали еще лучше.

Начнем с *внимания*. Мы надеемся, что сейчас вы читаете этот текст *внимательно*, обращая *внимание* на каждое предложение... но что это вообще значит? Хотя психологи изучают внимание давно, и многие его свойства хорошо известны, оказывается, что механизмы нашего привычного человеческого внимания до сих пор не вполне понятны, и исследователи до сих пор спорят о том, как оно устроено у людей. Еще в основополагающих работах Александра Лурии[1], заложившего основы современной нейропсихологии, модель работающего мозга делилась на (упрощенно говоря) три части, отвечающие за внимание, память и активацию коры головного мозга. Современные исследования показывают, что в формировании внимания большую роль играет рабочая память, в которой перерабатывается текущая информация (подобно оперативной памяти компьютера). А то, какая информация попадает в рабочую память, контролируется не только базовыми уровнями обработки входных сигналов, но и более высокоуровневыми когнитивными процессами [282]. Проще говоря, мы можем «обратить внимание» на что-то сознательным усилием... хотя о том, что такое «сознательное усилие», спорить можно еще дольше, чем о внимании.

[1] *Александр Романович Лурия* (1902–1977) — советский психолог, основатель отечественной и классик мировой нейропсихологии, развивший и продолживший идеи Льва Выготского. Основным его трудом стал двухтомник «Мозг и психические процессы» [591], который позднее был дополнен книгой «Основы нейропсихологии» [592]. В отличие от, к сожалению, многих советских ученых, Лурия был широко известен за рубежом, входил в академии наук разных стран; все его основные труды переводились и стали золотой классикой нейропсихологии.

В любом случае, речь в нейропсихологии идет скорее не об абстрактном «обратите внимание на вопрос», а о более конкретных механизмах внимания как фильтрации поступающей информации: на какой части поля зрения, каких звуках, запахах, ощущениях мы должны сконцентрироваться, то есть какую часть поступающей с органов чувств информации нужно передать в рабочую память и обрабатывать более сложными способами?

И для создателей искусственных нейронных сетей *внимание* относится к тому же самому вопросу: как выбрать из очень большого объема поступающих данных именно то, что нужно делать прямо сейчас? Как научить сверточную сеть «концентрироваться» на нужной части изображения при распознавании объектов, а рекуррентную сеть — на релевантной части предложения во время машинного перевода?

Начнем с внимания в обработке изображений. Очевидно, что решение проблемы внимания сильно улучшило бы результаты в таких задачах. Классические сверточные сети, о которых мы говорили в главе 5, отлично работают для распознавания объектов, но для того, чтобы найти какой-нибудь объект на картинке, им нужно перебрать тысячи возможных сегментов изображения, среди которых, возможно, найдется один нужный. Даже если изображения уже сжаты, и учитывая, что некоторые вычисления можно оптимизировать за счет переиспользования, больше всего ресурсов у большой сверточной сети уходит на применение сверточных фильтров ко всему изображению, то есть сложность в лучшем случае пропорциональна количеству пикселов.

С другой стороны, когда человек смотрит на рисунок, он не просматривает его во всех подробностях целиком. Мы уже обсуждали, что в основе своей фильтры сверточных сетей используют те же принципы, что и зрительная кора человека. Пикселы (фоторецепторы) объединяются в группы, которые реагируют на контраст, а эти группы в свою очередь выстраиваются дальше в более сложные иерархии, которые распознают линии и объекты.

Но еще одна, пока не использованная нами особенность человеческого зрения состоит в том, что оно активно использует механизм внимания [447]. Вместо того чтобы представлять все участки сцены в одинаковом виде с равной детализацией, мы фокусируем взгляд на отдельных небольших сегментах, которые (как нам кажется) содержат больше всего полезной информации. Подобный механизм внимания особенно полезен, если картинка шумная и содержит много «мусора». Новые модели машинного обучения используют эту идею и включают различные механизмы внимания в архитектуру нейронных сетей. Конечно, мы не напрямую копируем биологию (да и не понимаем мы механизмы внимания достаточно детально), а вдохновляемся ею.

В «классической» обработке изображений аналоги внимания появились достаточно давно. Мотивировались они в основном с точки зрения эффективности: хотелось заменить извечную концепцию скользящего окна, проходящего по всей картинке, на что-то более «разумное», сэкономив при этом массу вычислений на заведомо бессмысленных окнах. Отсюда происходит идея *каскадных моделей*, в которых отдельные классификаторы используются для того, чтобы последовательно уточнять расположение окон в изображении, на которые стоит посмотреть более

детально [155, 291, 552]. Другой аналогичный подход — распознавание только «интересных» частей изображения на основе простых статистик вроде перепадов контрастности или других низкоуровневых признаков [254].

Одним из первых примеров применения механизмов внимания в глубоких нейронных сетях стала работа [296], в которой Юго Ларошель (Hugo Larochelle) и Хинтон с помощью машин Больцмана третьего порядка моделировали механизмы фовеального зрения, при котором мы обращаем особое внимание на центральную часть зрительного поля, а периферийные участки поступают на вход размытыми, с меньшим разрешением. Идея была в том, чтобы позволить модели сделать только конечное небольшое число «фиксаций» на разных частях картинки: модель должна была обучиться распознавать изображения по этой неполной информации, а специальная часть модели, *контроллер*, должна была обучиться тому, куда, собственно, смотреть. Похожая архитектура была представлена и в [312].

Из этой идеи и происходят современные сети со вниманием. Быстро стало понятно, что важной составной частью механизма влияния является не только изначальный выбор частей изображения, на которые нужно посмотреть внимательно, но и *последовательность* этих выборов: глаз переходит от одной части изображения к другой не согласно заранее выбранному набору ключевых окон, а принимая решение каждый раз заново на основе уже накопленной информации. Поэтому модели с вниманием для обработки изображений — это тот естественный класс моделей, где сверточные сети объединяются с рекуррентными.

В работе исследователей из Google [441] была представлена одна из первых таких успешных моделей. Мы изобразили ее структуру на рис. 8.1:

- в каждый момент времени t на вход сети поступает предыдущее состояние h_{t-1} и произведенное из него положение l_t для нового «взгляда»;
- этот новый «взгляд» с помощью функции f_h преобразуется в вектор признаков g_t (от слова glimpse), который служит входом на шаге t;
- из h_{t-1} и g_t функцией f_h получается следующее скрытое состояние h_t;
- а из него уже получается собственно текущее «действие» $a_t = f_a(h_t)$ («действием» может быть, например, выдача ответа о том, какой объект удалось распознать) и положение следующего «взгляда» $l_{t+1} = f_l(h_t)$.

Задачи обучения таких моделей переносят нас в совершенно другие области, которых мы до сих пор не рассматривали. Например, модель на рис. 8.1 пытается решить задачу *обучения с подкреплением*, которому будет посвящена глава 9: сеть должна произвести следующее действие, то есть выбрать, но при этом функция ошибки (вознаграждение) получается не сразу, а только после того, как все «взгляды» закончатся, и модель выдаст собственно ответ в виде очередного a_t. Более того, оказалось, что базовый алгоритм обучения из [441] плохо масштабируется, и в последующей работе исследователей из Google DeepMind [15] подобная модель обучается с помощью вариационного вывода, до которого мы доберемся только в главе 10. Однако идея сетей с вниманием пригодится нам раньше, а детали обучения этих моделей все равно выходят за рамки нашей книги, поэтому мы говорим о сетях с вниманием именно сейчас и без особых подробностей.

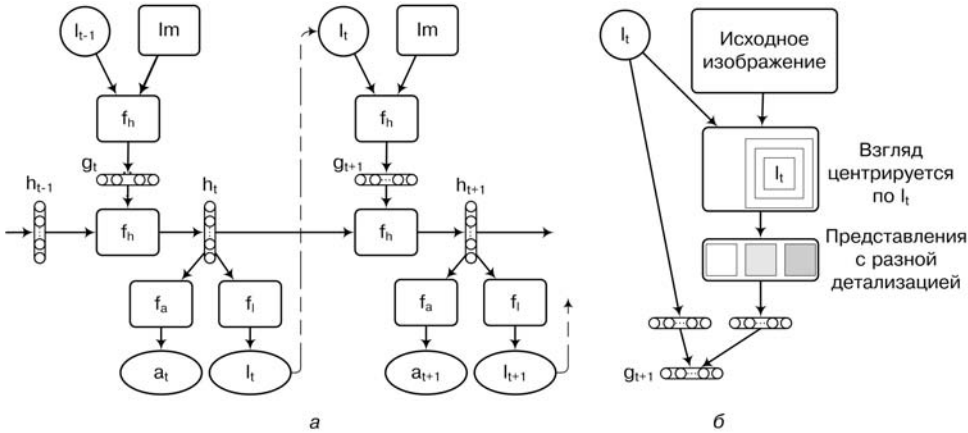


Рис. 8.1. Сеть с вниманием из [441]:
а — общая структура сети; *б* — обработка одного «взгляда»

В качестве первого примера рассмотрим интересное приложение: порождение подписей к картинкам. Интуитивно кажется, что механизм внимания должен давать здесь довольно много: когда мы переводим двумерную картинку в одномерное текстовое описание, было бы очень полезно понимать, какую именно часть картинки нужно описывать сейчас.

На рис. 8.2, *а* изображена структура одной из более ранних моделей для этого, описанной в статье под названием Show and Tell («Покажи и расскажи», популярное домашнее задание в младших классах американских школ) [495]. Это достаточно простая и прямолинейная модель, в которой по изображению строится его представление в виде вектора с помощью стандартной сверточной сети, в данном случае из [252]. А затем оно «разворачивается» в текст описания с помощью рекуррентной сети.

Статья, в которой в эту модель добавлен механизм внимания, вполне логично озаглавлена Show, Attend and Tell [496]. Эта архитектура изображена на рис. 8.2, *б*. Как и раньше, в модель добавляется новая сеть внимания f_{att}, которая на шаге t на основе текущего состояния порождающей подпись рекуррентной сети (то есть на основе выхода LSTM-ячеек) порождает значение внимания e_{ti} для каждой части изображения i с ее представлением $\boldsymbol{a}_i$ (мы считаем, что изображения разделены на не слишком большое число дискретных частей), а затем они нормализуются с помощью softmax, и получаются собственно веса внимания α_{ti}:

$$e_{ti} = f_{\text{att}}(\boldsymbol{a}_i, \boldsymbol{h}_{t-1}), \quad \alpha_{ti} = \frac{\exp(e_{ti})}{\sum_k \exp(e_{tk})}.$$

В работе [496] различаются два разных вида внимания:

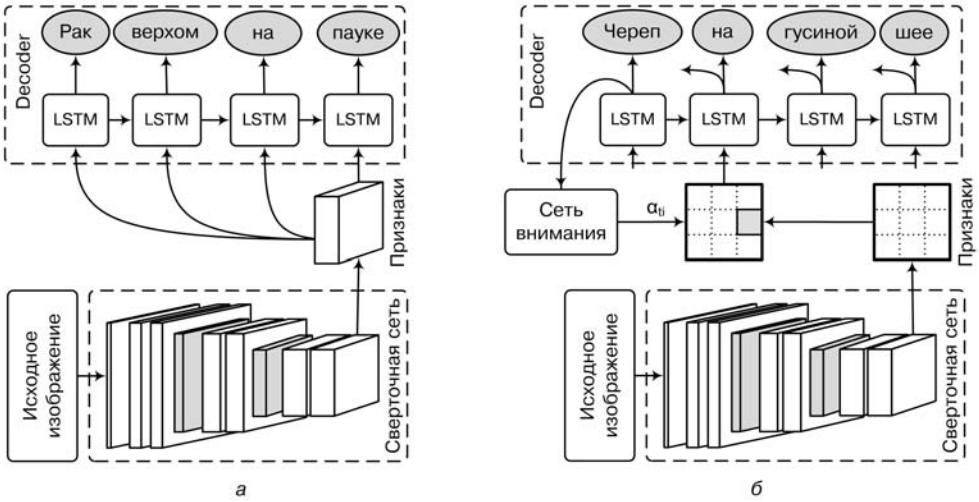


Рис. 8.2. Модели для порождения подписей к картинкам:
а — Show and Tell; *б* — Show, Attend and Tell

- *жесткое* (hard attention), в котором веса α_{ti} интерпретируются как вероятности событий s_{ti} того, что модель «посмотрит» в момент времени t на часть изображения i, и на вход рекуррентной сети для порождения текста подается представление той части изображения $\boldsymbol{a}_{i*}$, которая выпадет на кубике с вероятностями α_{ti};
- *мягкое* (soft attention), когда на вход рекуррентной сети подается, можно сказать, ожидание вектора из s_{ti}, $\sum_k \alpha_{ti}\boldsymbol{a}_i$, то есть модель в каждый момент «смотрит» на все части изображения, просто некоторые из них играют более важную роль, чем другие.

При этом модель с мягким вниманием по сути представляет собой самую обычную нейронную сеть, градиенты проходят через мягкое внимание беспрепятственно, и мы можем обучать всю модель целиком обычным градиентным спуском. А вот с жестким вниманием дела обстоят сложнее: трудно взять производную от броска кубика! Поэтому для обучения жесткого внимания в [496] используются вариационные приближения, о которых мы немного поговорим позже, в разделе 10.3.

В любом случае оба варианта умеют порождать вполне разумные подписи к фотографиям[1], и даже в случае ошибок механизм внимания позволяет достаточно

[1] По ряду соображений мы решили не копировать изображения из других статей, а показывать конструкцию и обучение моделей с вниманием на большом датасете в качестве практического примера в книге было бы слишком трудоемко, поэтому красивых картинок и порожденных к ним подписей прямо здесь не ждите. Однако они есть буквально в каждой из приведенных здесь ссылок.

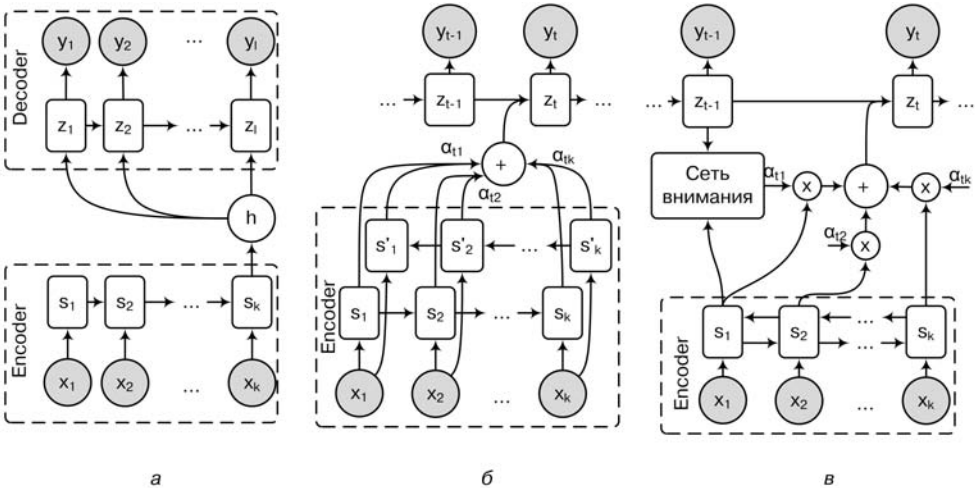


Рис. 8.3. Архитектуры encoder-decoder для машинного перевода: *а* — основная идея [269]; *б* — encoder-decoder с мягкой моделью выравнивания [22]; *в* — архитектура encoder-decoder с мягким вниманием [78]

легко понять, как именно ошиблась модель и на что она «смотрела» в момент ошибки, что очень важно для анализа ошибок.

Архитектура, которую мы сейчас рассмотрели, — это частный случай важной архитектуры *«кодировщик-декодировщик»* (encoder-decoder, и здесь мы, пожалуй, будем использовать англоязычное название), которая особенно часто используется в обработке естественного языка. Основная идея заключается в том, что мы уже видели на рис. 8.2, *а*: рекуррентные сети естественно использовать как вероятностные модели последовательностей [192], то есть мы можем обучить RNN, которая для $X = (x_1, x_2, \ldots, x_T)$ последовательно моделирует $p(x_1)$, $p(x_2 \mid x_1)$, …, $p(x_T \mid x_{<T}) = p(x_T \mid x_{T-1}, \ldots, x_1)$, а значит, и совместное распределение

$$p(X) = p(x_1)p(x_2 \mid x_1)\ldots p(x_k \mid x_{<k}\ldots p(x_T \mid x_{<T}).$$

Именно так рекуррентные сети применяются для языкового моделирования (language modeling) [28, 360], о котором мы говорили в главе 7: скрытое состояние используется как сжатое представление всей предшествующей истории, и следующее слово мы предсказываем из этого сжатого представления. Архитектура encoder-decoder всего лишь добавляет к этой вполне естественной идее еще одно условие: кодировщик сжимает некий вход в распределенное представление $\boldsymbol{z}$, и оно подается каждый раз на вход рекуррентной сети, которая работает декодировщиком (см. рис. 8.3, *а*, а также вспомните рис. 6.3, *г* — мы сейчас занимаемся в точности этим классом задач).

Одним из основных приложений таких моделей в обработке естественных языков стала задача *машинного перевода*; о ее постановке и некоторых проблемах с оценками качества мы уже говорили в разделе 7.1. Сейчас мы будем рассматривать машинный перевод в постановке выше, как задачу преобразования одной последовательности в другую, но стоит отметить, что нейронные сети используются и в других подходах к машинному переводу. Так, «классические» методы часто приближают $\log p(\boldsymbol{y} \mid \boldsymbol{x})$, где $\boldsymbol{x}$ — вход (предложение), а $\boldsymbol{y}$ — выход (его перевод), линейной комбинацией признаков, и нейронные сети успешно использовались как для переранжирования списков возможных переводов [476], так и, собственно, для обучения признаков [95, 141, 151, 349, 509, 520].

Ну а в такой постановке, как выше, идеально подходят как раз архитектуры encoder-decoder. Мы же не переводим слово за словом, а скорее строим для себя некое «семантическое представление», которое мы потом «разворачиваем» на другом языке. Эта интуиция в точности соответствует encoder-decoder архитектуре, как показано на рис. 8.3, *а*; к машинному переводу такая архитектура была применена в [309], и ее можно напрямую обучать на корпусе эталонных переводов предложений, как и делалось в [22, 269, 270, 309, 517].

Но у encoder-decoder архитектуры из [399] есть важный недостаток: все предложение целиком приходится «сворачивать» в вектор фиксированной размерности. Это значит, что длинное предложение будет гораздо сложнее свернуть и развернуть, чем короткое. И действительно, практика показывает, что с увеличением длины входа качество падает радикально; сильно увеличивать размер скрытого представления тоже не получается, потому что тогда модель становится слишком большой, и ее не получается обучать. В некоторых работах эту проблему пытались решать автоматической сегментацией [406], но оказалось, что механизм внимания работает еще лучше.

В машинном переводе внимание помогает понять, какую именно часть входа мы сейчас переводим. В работе [22] мягкая модель выравнивания (soft alignment model) выдает веса α_{ti}, которые показывают, насколько каждое из слов входа влияет на слово, которое мы сейчас переводим. Скрытые представления каждого слова при этом получаются обычным двусторонним LSTM, который строит левый и правый контекст каждого слова в предложении, и всю модель можно обучить одновременно, градиентным спуском (см. рис. 8.3, *б*).

А начиная с работы [78], машинный перевод делается с мягким механизмом внимания (soft attention), как показано на рис. 8.3, *в*: дополнительная маленькая нейронная сеть получает на вход текущее скрытое состояние декодировщика и локальное представление слова и выдает оценку релевантности α_{ti} для каждого слова. И снова эту модель можно просто обучать градиентным спуском на датасете параллельных переводов, целиком (end-to-end), без каких-то дополнительных ухищрений и многоэтапных процессов.

Это очень важная особенность современного машинного обучения, особенно обучения глубоких сетей: как видите, попробовать даже совершенно новую архитектуру обычно очень просто, и требуются скорее инженерные усилия (выбрать

правильную структуру компонентов сети, подобрать параметры, найти верное расписание оптимизации), чем математические. А в тех редких случаях, когда так не получается, обычно удается построить вариационную оценку и оптимизировать ее; об этом мы поговорим в разделе 10.3.

Механизмы внимания для перевода продолжают развиваться: чтобы решить важную проблему редких слов, в [3] вводится дополнительная модель выравнивания слов, которая строит соответствия между конкретными словами так, чтобы можно было просто посмотреть редкие слова в словаре; в [401] базовую модель внимания дополняют выборкой по значимости (importance sampling), что позволяет расширить ее на очень большие словари; в [161] строится многоязыковая модель, число параметров в которой растет линейно по мере добавления новых языков; в [81] модель машинного перевода начинает работу от отдельных символов, и так далее (см. также обзор разных архитектур внимания для машинного перевода в [341]).

Но, может быть, все это сухая теория, а на практике люди просто строят большие базы данных хороших переводов? Оказывается, нет: важная недавняя статья от Google [187] показывает, как работает система Google's Neural Machine Translation (GNMT), то есть фактически Google Translate. И, к удовольствию исследователей, оказалось, что Google делает ровно то же самое: Google Translate состоит из кодировщика, декодировщика и сети внимания. Однако есть и несколько интересных новых трюков и отличий:

- рекуррентные сети должны быть достаточно глубокими, чтобы суметь выразить разные нерегулярности и особенности естественных языков, так что в GNMT используются по восемь уровней LSTM в кодировщике и декодировщике;
- но, как мы уже обсуждали в главе 6, просто ставить друг на друга уровни LSTM не очень помогает, архитектуры глубже трех-четырех уровней так обучить очень сложно; поэтому в GNMT добавляются остаточные связи между уровнями, точно как в ResNet-подобных архитектурах, о которых мы говорили в разделе 5.4;
- нижний слой при этом, естественно, двунаправленный, чтобы не потерять контекст слова как слева, так и справа;
- а еще в GNMT есть два особых трюка, которые могут разбивать слова на части и переводить фрагментами; любопытно, что эти фрагменты не обязательно представляют собой отдельные буквы, и модель сегментации слов изначально происходит из модели сегментации для азиатских языков (разбить китайский текст на слова — непростая задача), которая, оказывается, помогает и в сугубо европейском переводе.

Если вы активно пользуетесь Google Translate, вы наверняка заметили переход к системе GNMT: во второй половине 2016 года в какой-то момент переводы действительно стали заметно лучше; это и был эффект GNMT.

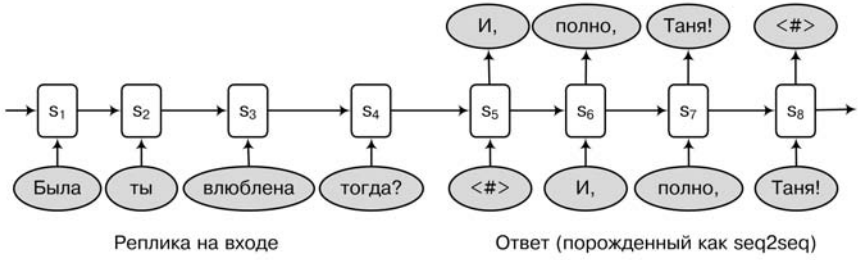


Рис. 8.4. Диалоговая модель на основе архитектуры seq2seq [551]

Обратите внимание, что до сих пор мы говорили о том, как перевести одно предложение. Никакая дополнительная сеть с вниманием не позволит перевести «Войну и мир» целиком. А чтобы следующие предложения как-то зависели от предыдущих, нужно сохранять контекст. Это важно для перевода, но становится еще важнее в *диалоговых моделях*, которые пытаются поддерживать разговор с человеком, ведь очередная реплика в диалоге часто совсем не информативна, и за подходящим контекстом нужно обращаться к более далекому прошлому.

Можно сказать, что современные диалоговые модели, основанные на нейронных сетях, начались с работы [551], в которой поддержание диалога рассматривается как *seq2seq*-задача [517]. По сути это все та же архитектура encoder-decoder: мы порождаем ответ, используя свернутую входную реплику в качестве контекста, как показано на рис. 8.4. Однако, хотя результаты в [551] и для диалогов о конкретной предметной области, и даже для общечеловеческих разговоров «по душам» получаются вполне разумными[1], такая модель все еще совсем никак не обрабатывает глобальный контекст диалога, не сохраняет его содержание от реплики к реплике.

Поэтому в работе [214] к диалоговым системам была применена архитектура *иерархического кодировщика-декодировщика* (hierarchical recurrent encoder-decoder architecture, HRED), которую ранее применяли в [215] для контекстно-зависимых подсказок в системах информационного поиска (то, что появляется под поисковой строкой, когда вы начинаете в ней печатать). Основная идея работы [214] состоит в том, чтобы рассматривать диалог как двухуровневую систему, последовательность высказываний, каждое из которых — это последовательность слов; для моделирования такой системы HRED обучает:

- *кодировщик* (encoder RNN), рекуррентную сеть, которая «сворачивает» каждое высказывание собеседника в вектор фиксированной размерности;

[1] Наборы данных для диалоговых систем — это обычно или диалоги с техподдержкой, которые связаны с какой-то конкретной областью, например Ubuntu, или датасет «общего назначения» OpenSubtitles, содержащий субтитры к фильмам и сериалам; кроме примеров достаточно естественных диалогов, субтитры еще и дают отличный параллельный корпус для машинного перевода, потому что фильмы часто снабжены субтитрами на разных языках [531].

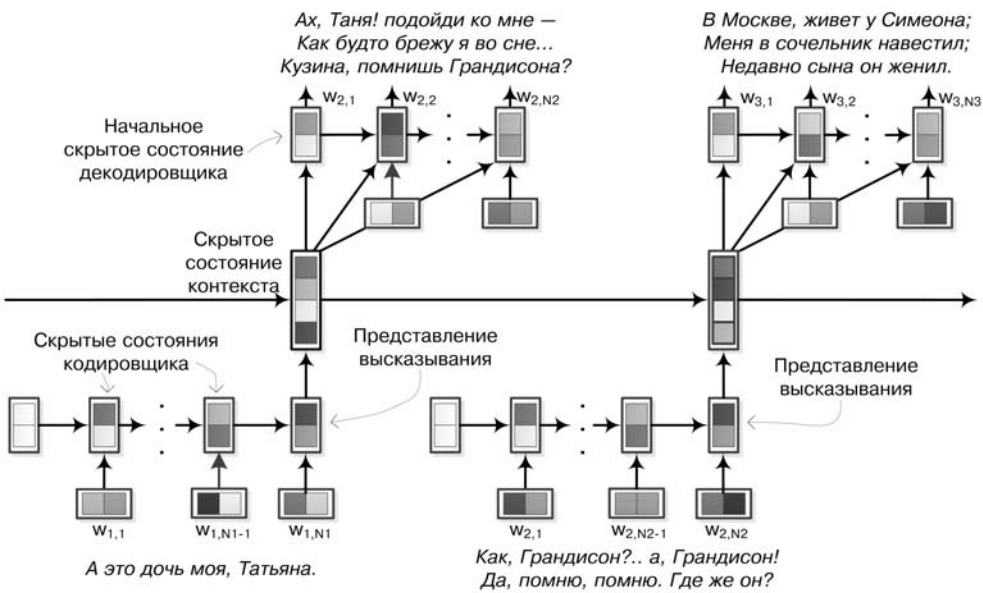


Рис. 8.5. Диалоговая модель на основе иерархической encoder-decoder архитектуры (HRED) [214]

- *контекстную сеть* (context RNN), которая последовательно (и тоже рекуррентно) обрабатывает все предыдущие высказывания и выдает текущий вектор контекста;
- *декодировщик* (decoder RNN), сеть, которая последовательно предсказывает слова для следующего высказывания системы при условии текущего контекста.

Все это великолепие показано на рис. 8.5. В [214] используются двунаправленные RNN, веса инициализируются *word2vec*-представлениями, обученными на большом датасете (Google News), первая фаза обучения делается на корпусе субтитров, а затем основное обучение происходит на большом датасете *MovieTriples*, который состоит из несложных вопросов и ответов, основанных на датасете субтитров *Movie-DiC* [29]; результаты получаются достаточно многообещающие, и подход, основанный на HRED-архитектуре, активно развивается и по сей день. Например, в [212] для HRED строится вариационная нижняя оценка, что позволяет добавить скрытые переменные, моделирующие зависимости между репликами. Сети с вниманием также важны для диалоговых систем: например, в модели Attention with intention («Внимание с намерением») [575] отдельная сеть моделирует намерения участников диалога (например, намерение сообщить ту или иную информацию).

Хочется подвести итог разделу, посвященному сетям с вниманием, но с итогами здесь сложно: слишком уж все пока что в процессе, слишком быстро развивается это направление. Так, работа с характерным названием *Attention Is All You Need* [12] доводит идею сетей с вниманием до логического завершения: там строится архитектура сети для машинного перевода, в которой вообще нет рекуррентных компонентов! Все делается исключительно на многомерных картах внимания; авторы утверждают, что получают результаты не хуже, а то и лучше, чем архитектуры с рекуррентными кодировщиком и декодировщиком, и при этом сеть обучается в десятки раз быстрее. В целом, сейчас механизмы внимания привлекают, простите за каламбур, все больше внимания исследователей, и не исключено, что в будущем роль механизмов внимания в нейронных сетях станет еще важнее.

8.2. Порождающие модели и глубокое обучение

> Есть проповедники смерти; и земля полна теми, кому нужно проповедовать отвращение к жизни... «Трудно рожать, — говорят другие, — к чему еще рожать? Рождаются лишь несчастные!» И они также проповедники смерти.
>
> *Ф. Ницше. Так говорил Заратустра*

До сих пор мы рассматривали задачи, в которых была хорошо определенная целевая функция. Да и вообще, если честно, все это время мы в основном рассматривали задачи классификации: как взять большой и сложный вход высокой размерности, постепенно выделить в нем интересные признаки, глубокими моделями довести дело до совсем уж «умных» признаков... но затем все равно вернуть модель на землю, преобразуя результат в распределение на относительно маленьком числе классов. Да и что нам было делать, если целевая функция в имеющихся наборах данных действительно обычно представляет собой ошибку классификации.

Однако это не единственный класс задач, который хотелось бы научиться решать. В этой главе мы рассмотрим один из возможных ответов на другой интересный вопрос: как научиться *порождать* новые объекты, похожие на объекты из данных? Например, как на основе датасета MNIST научиться порождать рукописные цифры? Даже если у вас обучился самый лучший классификатор на свете, как попросить его сгенерировать новые варианты цифр? Можно, например, начать с последнего слоя и попробовать найти входы, которые сильно активируют нейроны, соответствующие той или иной цифре на выходе. Люди делают так при анализе сверточных сетей, и в результате действительно получается нечто более или менее разумное, что может хорошо визуализировать активации каждого отдельного нейрона [585]. Но на хороший рукописный генератор эти результаты никак не тянут. Да и ладно рукописные цифры, эту задачу уже со всех сторон разбили, но что если мы захотели бы генерировать более сложные изображения, например фото интерьеров или изображения человеческих лиц, как в [432]?

Говоря на языке математики, сейчас мы приходим к разнице между *дискриминативными*, разделяющими моделями и *генеративными*, порождающими:

- дискриминативные модели обучают функцию, которая отображает вход $\boldsymbol{x}$ в некоторую метку класса y; в вероятностных терминах это значит, что они обучают *условное* распределение $p(y \mid \boldsymbol{x})$;
- порождающие (генеративные) модели обучают *совместное* распределение данных $p(\boldsymbol{x}, y)$; это можно использовать для того, чтобы получить $p(y \mid \boldsymbol{x})$, ведь для фиксированного $\boldsymbol{x}$ мы получим просто $p(y \mid \boldsymbol{x}) = \frac{p(\boldsymbol{x},y)}{p(\boldsymbol{x})} \propto p(\boldsymbol{x},y)$, но совместное распределение дает больше информации, его можно использовать, например, для порождения новых данных.

Еще одно вытекающее из этого важное различие состоит в том, что дискриминативные модели решают только задачи обучения с учителем, а порождающие модели могут пытаться решать и задачи обучения без учителя, когда меток никаких нет, и нужно просто смоделировать распределение данных $p(\boldsymbol{x})$; поэтому во многих практических задачах часто бывают нужны именно порождающие модели[1].

С математической точки зрения основная цель порождающей модели обычно состоит в максимизации функции правдоподобия: для набора данных $D = \{\boldsymbol{x}_i\}_{i=1}^N$ максимизировать $\prod_{i=1}^N p(\boldsymbol{x}_i; \boldsymbol{\theta})$ по параметрам модели, то есть после логарифмирования искать

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i; \boldsymbol{\theta}).$$

Важен и другой взгляд на то же самое: максимизация правдоподобия эквивалентна минимизации расстояния Кульбака — Лейблера между распределением p, которое получается из нашей модели, и распределением $\hat{p}_{\text{data}}$ — эмпирическим распределением данных. Это эмпирическое распределение попросту полностью сосредоточено в точках из датасета и равномерно распределено по ним, так что:

$$\mathrm{KL}\left(\hat{p}_{\text{data}}(\boldsymbol{x}), p(\boldsymbol{x}; \boldsymbol{\theta})\right) = \int \hat{p}_{\text{data}}(\boldsymbol{x}) \log p(\boldsymbol{x},\boldsymbol{\theta}) \mathrm{d}x = \sum_{i=1}^{N} \hat{p}_{\text{data}}(\boldsymbol{x}_i) \log p(\boldsymbol{x}_i,\boldsymbol{\theta})$$

и минимизация этого выражения эквивалентна максимизации того, что выше. Иначе говоря, точки данных, в которых сосредоточено распределение $\hat{p}_{\text{data}}$, «тянут вверх» распределение p.

Но откуда взять распределение $p(\boldsymbol{x},\boldsymbol{\theta})$ и как использовать для него нейронные сети? Генеративные модели различаются как раз тем, как именно они строят распределение $p(\boldsymbol{x},\boldsymbol{\theta})$. Можно строить это распределение *явно*, делая вероятностные предположения, которые обычно сводятся к тому, что общее распределение $p(\boldsymbol{x},\boldsymbol{\theta})$ выражается в виде произведения тех или иных «маленьких» распределений.

[1] См. также обсуждение порождающих моделей в [185], которому мы будем отчасти следовать в этом разделе.

Например, *байесовские сети доверия* строят распределение из условных распределений вида $p(x_i \mid x_{j_1}, \ldots, x_{j_k})$ для каждого i:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i \mid X_i).$$

В этих моделях вероятностные предположения об условной зависимости и независимости между переменными выражаются в виде того, какие переменные входят в X_i [416, 594].

Можно даже и вовсе никаких предположений не делать: любое распределение всегда раскладывается так:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1}).$$

Если теперь моделировать все эти условные вероятности последовательно глубокими нейронными сетями, получится модель, которая сможет последовательно породить $\boldsymbol{x}$ компонент за компонентом, каждый раз для порождения x_i опираясь на уже порожденные $x_1, \ldots, x_{i-1}$.

Другой подход к порождению сложных распределений, идею которого мы будем использовать и в состязательных сетях, состоит в том, чтобы начать с простого распределения $p(\boldsymbol{z})$ на скрытые факторы $\boldsymbol{z}$ и затем применить к нему сложное преобразование, которое и будет содержать в себе всю сложность требующихся многообразий. Обычно в качестве $p(\boldsymbol{z})$ можно взять обычное нормальное распределение, а задача состоит в том, чтобы обучить биективную функцию $f : X \to Z$ так, чтобы распределение данных на X превращалось бы в простое распределение на Z. Тогда, чтобы сгенерировать новую точку из распределения, похожего на p_{data}, достаточно будет породить точку из гауссиана, а затем применить обратную функцию $f^{-1} : Z \to X$. А обучить такую f можно опять просто максимизируя правдоподобие, ведь через непрерывные функции f градиент прекрасно «протаскивается» с помощью формулы замены переменных:

$$p_X(\boldsymbol{x}) = p_Z(f(\boldsymbol{x})) \left| \det \left(\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right) \right|^{-1},$$

где $\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$ — это матрица частных производных (якобиан) функции f. Конечно, считать якобиан «в лоб» было бы слишком сложно, но часто можно что-то придумать. Например, в работе [121] предложена специальная форма функции f, для которой определитель подсчитать легко (якобиан оказывается треугольным), а составные части этой функции можно затем моделировать чем угодно — например, нейронными сетями. В [121] этот подход предлагается для порождения изображений, и результат получается довольно убедительный.

Отчасти мы начали отвечать на эти вопросы, когда рассматривали рекуррентные нейронные сети. Рекуррентные архитектуры обычно обучаются предсказывать следующий элемент последовательности, и поэтому их естественным образом можно приспособить к порождению, как мы делали с языковыми моделями. Но это пока не отвечает на наш вопрос о порождении рукописных цифр, ведь последовательности как таковой здесь нет, и как их использовать, пока неясно[1]. Мы ответим на этот вопрос чуть позже, а пока продолжим о порождающих моделях в целом.

Для чего нужны порождающие модели? Во-первых, они позволяют нам проверить, насколько хорошо мы поняли распределение данных; глядя на активации фильтров обычного автокодировщика, понять это нелегко, а из порождающей модели можно, собственно, что-нибудь породить и посмотреть, как получается.

Во-вторых, порождающая модель иногда позволяет использовать не только размеченные данные, делая обучение с частичным привлечением учителя (semi-supervised learning). Если вы хотите отличить кошек от собак на фотографиях, у вас может быть не так уж много хорошо размеченных данных, на которых кошки и собаки старательно отмечены вручную. Но в любом случае львиная доля задачи состоит в том, чтобы вообще понять, чем разумные фотографии отличаются от случайного шума в миллионах пикселов. Иначе говоря, распределение $p(y \mid \boldsymbol{x})$, в котором y — это один бит «котик ли это?», а $\boldsymbol{x}$ — целая фотография, может быть проще обучить, если сначала узнать что-то о распределении $p(\boldsymbol{x})$ в целом.

В-третьих, порождающие модели хорошо справляются с задачами, в которых может быть несколько правильных ответов. Представьте себе, например, задачу предсказания следующего кадра в видеоролике [333]. Действие может развиваться несколькими разными способами, и все они являются примерно одинаково хорошими ответами на вопрос о том, каким будет следующий кадр. Однако если взять и усреднить все возможные ответы, получится что-то смазанно-промежуточное, а разумного кадра из видео как раз не выйдет. А порождающая модель должна быть способна обучить *мультимодальное* распределение, то есть распределение с несколькими пиками, соответствующими нескольким возможным ответам.

Ну и, наконец, в-четвертых: иногда просто действительно нужно именно порождать ответы. Например, предположим, что мы хотим сгенерировать новую фотографию с милым котиком на ней. Может показаться, что здесь нам помогут конструкции автокодировщиков из раздела 5.5: в любом автокодировщике была декодирующая часть, которая по набору скрытых факторов способна восстановить исходного котика. Но оказывается, что напрямую использовать эту часть как порождающую сеть не удается: откуда взять подходящий набор скрытых факторов? С чего начинать восстановление входа?

---

[1] Отметим здесь мельком, что рекуррентные архитектуры действительно применяются к обработке изображений; существует целый ряд работ, посвященных так называемым рекуррентным сверточным сетям (recurrent convolutional networks) [325, 415, 425, 484], однако все эти модели скорее просто помогают улучшить сверточные признаки и не приводят к тому, чтобы научиться хорошо порождать изображения.

Рис. 8.6. Таксономия порождающих моделей [185]

Да, автокодировщик может спроецировать многообразие данных из высокой размерности в гораздо более низкую, но в этой низкой размерности многообразие все равно остается довольно сложным, и нельзя просто взять и подобрать такие скрытые факторы, чтобы из них получался разумного вида котик. Конкретно этому горю мы поможем в разделе 8.5, а пока продолжим о порождающих моделях в целом.

Общая таксономия порождающих моделей в контексте глубоких нейронных сетей, предложенная Иэном Гудфеллоу в [185], показана на рис. 8.6. В книге мы не будем подробно останавливаться на каждой из этих конструкций; ограничимся парой примеров, а затем сразу перейдем к основному содержанию: порождающим состязательным сетям.

Основной критерий, по которому различаются порождающие модели — это то, представляют ли они плотность $p(\boldsymbol{x},y)$ в явном виде; иначе говоря, можем ли мы подсчитать $p(\boldsymbol{x},y)$ как функцию от $\boldsymbol{x}$ и y или модель представляет собой черный ящик, который может генерировать новые примеры $(\boldsymbol{x},y)$, но считать плотность не умеет.

Как правило, модели, где плотность известна явно, делают какие-то дополнительные предположения на структуру этих распределений. Или делают их позже, как, например, появившиеся еще в 1990-х годах FVBN (fully visible belief networks) [164], в которых плотность распределения модели представляется так:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i \mid x_1 \ldots, x_{i-1}).$$

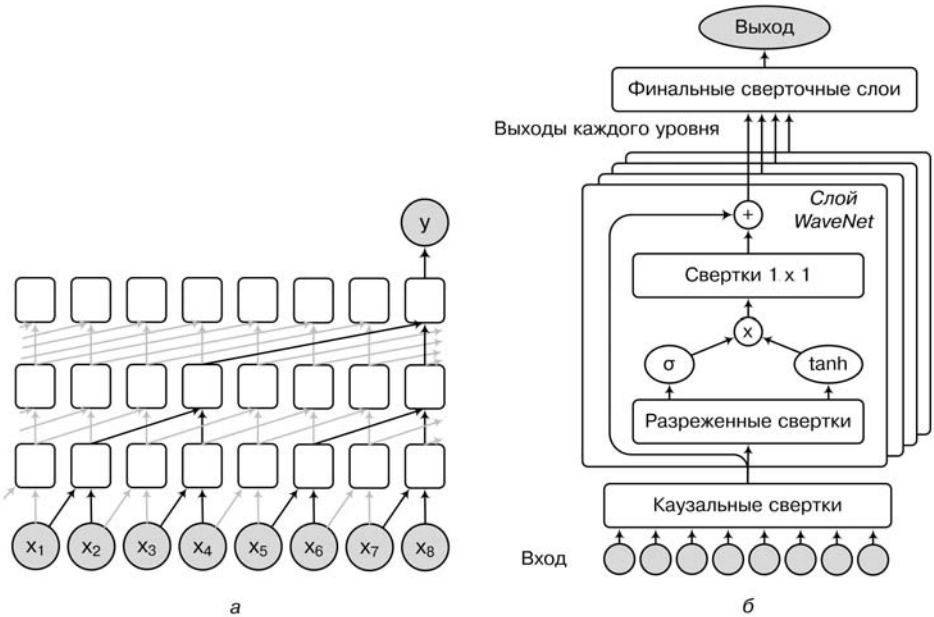


Рис. 8.7. WaveNet: *а* — каузальные свертки; *б* — структура модели [560]

Такое разложение верно всегда, и упрощающие предположения здесь начинаются позже, когда мы моделируем одномерные распределения $p(x_i \mid x_1 \ldots, x_{i-1})$. Идея FVBN состоит в том, что с одномерными распределениями мы уж как-нибудь разберемся — в ранних работах их представляли теми или иными классическими моделями машинного обучения, а сейчас мы можем их промоделировать нейронной сетью.

Именно эта идея лежит в основе одной из лучших в настоящий момент моделей для работы со звуком, WaveNet, разработанной во все той же компании Google DeepMind [560]. Мы мало говорили о звуке в этой книге, да и работ, где для порождения звука использовались бы глубокие нейронные сети, не так много, поэтому кратко расскажем о WaveNet. Идея, приходящая из FVBN, здесь состоит в том, чтобы моделировать условное распределение

$$p(\boldsymbol{x} \mid \mathbf{h}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}, \mathbf{h})$$

с помощью нейронных сетей. На звуковых данных полезно делать одномерные свертки, но свертки не должны «забегать вперед» по времени, поэтому в WaveNet используются так называемые *каузальные свертки* (causal convolutions), которые

смотрят только назад. Кроме того, их разумно прореживать со временем, чтобы получался «обобщенный» взгляд на более далекую историю. Все это изображено на рис. 8.7, *а*, где черные стрелочки показывают связи, участвующие в порождении очередного выхода: обратите внимание, что все входы $x_1, \ldots, x_8$ оказывают влияние на выход третьего сверточного слоя, но при этом каждый вход за счет разреженной структуры более поздних слоев участвует только один раз. А дальше с помощью таких сверток строится архитектура, изображенная на рис. 8.7, *б*. Эта архитектура состоит из нескольких последовательных слоев разреженных сверток, управляемых похожей на гейты структурой, и в ней снова встречаются трюки, которые мы уже не раз видели: остаточные связи, связи «через уровень» и так далее. В результате порождать речь получается довольно хорошо, да и в порождении музыки это, наверное, одна из лучших существующих моделей[1].

Похожая идея последовательного порождения объекта в зависимости от уже порожденной его части развивается в моделях PixelRNN [548] и PixelCNN [94], в которых модель строит изображение пиксел за пикселом, слева направо и сверху вниз. Каждый очередной пиксел x_n порождается из условного распределения $p(x_n \mid x_1, \ldots, x_{n-1})$, а оно уже моделируется или рекуррентной сетью, как в PixelRNN, или сверточной, как в PixelCNN. Как и в случае WaveNet, в обоих случаях приходится внести некоторые изменения в структуру рекуррентной и сверточной сетей. В подробности мы здесь вдаваться не будем, но главные особенности PixelRNN и PixelCNN изображены на рис. 8.8: на рис. 8.8, *а* показан общий вид сверточных фильтров в PixelCNN, в которых специальная маска делает так, чтобы сеть не могла «забежать вперед» при последовательном порождении, а на рис. 8.8, *б* представлены связи в двух вариантах PixelRNN: построчный вариант подает на вход LSTM признаки из предыдущего слоя пикселов, подсчитанные сверточной архитектурой, а двунаправленный диагональный LSTM старается использовать весь накопившийся контекст. Есть и похожие модели с вниманием (см. раздел 8.1), которые в известной работе [15] применили к распознаванию образов, а недавно успешно применили и к их порождению [127]. Модель DRAW из [127] последовательно «рисует» картинку с помощью рекуррентной сети, а механизм внимания помогает сети в данный момент сконцентрироваться на нужной части изображения.

Если хочется явно выразить совсем сложные распределения в порождающих моделях, их приходится приближать более простыми, которые уже, в свою очередь, могут быть выражены явно. Для этого обычно используются вариационные методы, о которых мы кратко поговорим в разделе 10.3.

Основная альтернатива всему этому состоит в том, чтобы использовать *неявные* (implicit) порождающие модели, в которых мы не пытаемся получить функцию, подсчитывающую плотность нужного распределения в каждой точке, а просто моделируем то, что нам от этой модели нужно. А нужно обычно уметь *сэмплировать*

[1] Результаты WaveNet можно услышать на сайте DeepMind [547].

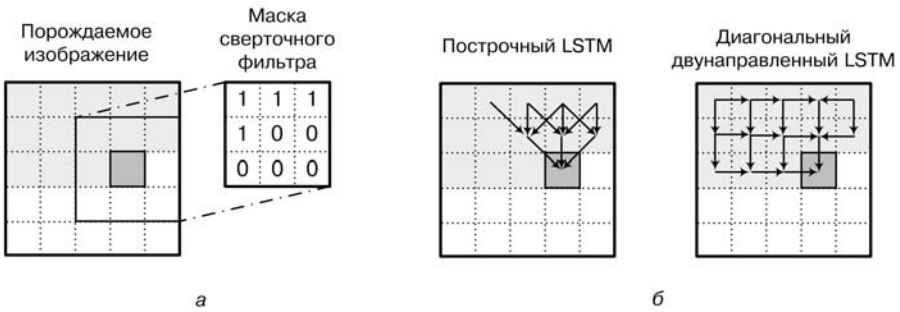


Рис. 8.8. Порождаем картинку пиксел за пикселом: *а* — PixelRNN; *б* — PixelCNN

из распределения, то есть, собственно, порождать новые точки по этому сложному распределению. Например, если мы хотим просто научиться порождать фотографии милых котиков, нам не так важно иметь явную функцию плотности $p(\boldsymbol{x})$, которая могла бы сказать, насколько вероятно, что перед нами котик, — вполне достаточно просто уметь генерировать новые $\boldsymbol{x} \sim p(\boldsymbol{x})$.

В классическом байесовском выводе сэмплирование из сложных многомерных распределений делается с помощью марковских цепей: попробуем построить марковскую цепь, которая описывает случайное блуждание под графиком плотности распределения; если достаточно долго блуждать под графиком плотности $p(\boldsymbol{x})$, можно будет считать, что полученная точка представляет собой случайную точку, взятую по распределению $p(\boldsymbol{x})$. Такой подход называется MCMC-методами, от слов *Markov chain Monte Carlo* [44, 250]. Полноценное описание этих методов выходит далеко за рамки нашей книги. Отметим только, что уже предпринимались вполне успешные попытки моделировать эту марковскую цепь глубокой нейронной сетью; результат известен как *порождающие стохастические сети* (Generative Stochastic Networks) [17, 41, 199]. Но мы с вами в этой главе пойдем другим путем...

8.3. Состязательные сети

> Я смотрю на него [Месси] не как на соперника, а как на человека, который делает меня лучше, а я делаю лучше его.
>
> *К. Роналду*

«Настоящие» порождающие сети требуют довольно сложных алгоритмов вывода, такие модели часто получаются довольно хрупкими, и обучать их — это совершенно отдельная, довольно сложная наука. Но есть и другие идеи, которые тоже позволяют весьма удачно реализовать порождающие модели и при этом обходятся без всей сложной машинерии генеративных моделей без учителя. Например, в главе 10

мы рассмотрим вариант обучения вероятностной модели с помощью вариационных приближений, которые пытаются построить наилучшее приближение к сложному апостериорному распределению, выбирая его из (очень богатого) класса распределений, порождаемых нейронной сетью. На выходе получится сеть, которая может генерировать новые примеры из хорошего приближения к апостериорному распределению.

А сейчас мы подробно рассмотрим одну из моделей, идея которых еще проще, но при этом они прекрасно работают и сейчас находятся на переднем крае в области изучения нейронных сетей, — *порождающие состязательные сети* (Generative Adversarial Networks, GANs)[1]. Основная идея действительно очень проста, но появилась совсем недавно, в работе 2014 года, главным автором которой был Иэн Гудфеллоу, аспирант Йошуа Бенжи [176]. А сейчас порождающие состязательные сети развиваются очень быстро, и, как пишет в одном из своих постов Ян Лекун [77], активно используются в *Facebook* для обработки изображений и видеороликов.

Начнем с того, что же и с чем в GAN состязается. Множественное число в названии модели на самом деле неспроста: в базовом варианте модель порождающих состязательных сетей состоит из двух искусственных нейронных сетей, которые соперничают друг с другом. Одна из них, *генератор* (generator; все связанное с генератором далее будет обозначаться буквой g или G), порождает объекты в пространстве данных, а вторая, *дискриминатор* (discriminator; про него мы будем говорить с индексами d или D), учится отличать порожденные генератором объекты от настоящих примеров из обучающей выборки. Таким образом, получается, что модель GAN состоит из двух частей с противоположными целями:

- цель дискриминатора — доказать, что генератор творит какую-то ерунду, научившись надежно отличать порожденные генератором примеры от настоящих; иначе говоря, дискриминатор решает самую обычную задачу бинарной классификации: по заданному примеру, выглядящему как элемент пространства данных, решить, был ли он «настоящим» или был порожден генератором;
- а цель генератора — «обмануть» дискриминатор, сделать так, что дискриминатор не сможет различить распределение данных p_{data} и распределение p_{gen}, которое порождает генератор; если бы дискриминатор работал идеально, то эта цель совпадала бы с целью научиться порождать данные из в точности такого распределения, как во входной выборке; на практике получается не настолько идеально, но все равно неплохо.

[1] Два замечания о терминологии. Во-первых, иногда по-русски GAN называют «конкурирующими сетями», но мы против: конкуренция должна быть *за* что-то, или по крайней мере между сущностями, которые пытаются делать одно и то же; а здесь задачи у сетей прямо противоположные, примерно как в состязательном судебном процессе. Во-вторых, мы сначала хотели было пользоваться для «порождающих состязательных сетей» русскоязычной аббревиатурой, но самой цензурной ассоциацией на нее у нас оказалось полное собрание сочинений Владимира Ильича Ленина, так что пусть уж аббревиатура будет английской; это и в поиске вам поможет, если вдруг захотите узнать о GAN'ах побольше.

Видите, как все просто? Одна сеть пытается научиться порождать правильные примеры, обманывая вторую, а вторая пытается отличить порожденные первой примеры от настоящих. По мере обучения они постепенно делают друг друга лучше, как принципиальные соперники в спорте. Практическая цель всего этого состоит в том, чтобы генератор в конце концов победил и научился таки делать p_{gen} настолько похожим на p_{data}, что никто не отличит — но чтобы побеждал он обязательно в сложной честной борьбе, иначе настоящим чемпионом ему не стать.

Давайте теперь формализуем GAN. Для этого нужно понять, каким образом нейронная сеть может работать генератором, как она может порождать объекты из заданного распределения. Это не вполне очевидно: нейронная сеть умеет переводить входы в выходы, но не имеет сама по себе никакой возможности генерировать новые примеры из ничего. Кроме того, нам хотелось бы, чтобы сеть порождала разные новые примеры, а не просто запомнила несколько готовых вариантов из обучающей выборки. Поэтому вместо того, чтобы генерировать с нуля, нейронная сеть будет *преобразовывать* случайные входы, сгенерированные из некоторого «посевного» распределения, которое во всех наших примерах будет просто стандартным нормальным распределением $\mathcal{N}(0,1)$ (но многомерным, конечно). То есть вместо того, чтобы просить сеть порождать примеры, что ей несвойственно, мы будем просить ее преобразовать простую функцию (стандартное нормальное распределение) в сложную (распределение данных) — а это как раз задача, для которой нейронные сети отлично подходят. Кстати, мы еще встретимся с такой архитектурой в главе 10: вариационный автокодировщик будет управляться с аппроксимацией распределения ровно тем же способом.

Поэтому формально генератор можно записать так:

$$G = G(\boldsymbol{z}; \boldsymbol{\theta}_g) : Z \to X,$$

где Z — некоторое пространство скрытых (латентных) факторов, на котором задано априорное распределение $p_z(\boldsymbol{z})$. А дискриминатор, в свою очередь, выглядит так:

$$D = D(\boldsymbol{x}; \boldsymbol{\theta}_d) : X \to [0,1].$$

Он отображает объекты из пространства данных в отрезок $[0,1]$, который интерпретируется как вероятность того, что пример был действительно «настоящий», из p_{data}, а не сгенерированный из p_{gen}. Цель дискриминатора состоит в том, чтобы на обучающей выборке выдавать максимальный результат, а на порожденных генератором примерах — минимальный.

Целевая функция для дискриминатора здесь лежит буквально на поверхности: мы бы хотели, чтобы на примерах из p_{data} ожидаемый ответ дискриминатора был как можно больше, а на примерах из p_{gen} — как можно меньше, то есть дискриминатор хочет максимизировать следующую величину:

$$\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}\left[\log D(\boldsymbol{x})\right] + \mathbb{E}_{\boldsymbol{x}\sim p_{\text{gen}}(\boldsymbol{x})}\left[\log(1 - D(\boldsymbol{x}))\right],$$

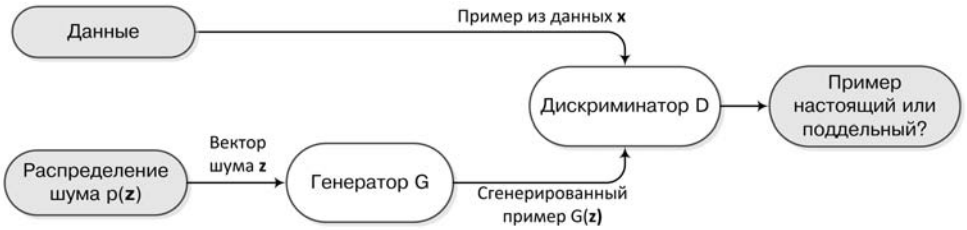


Рис. 8.9. Схема работы порождающей состязательной сети (GAN)

где $p_{\text{gen}}(\boldsymbol{x})$ — это порождаемое генератором распределение, $p_{\text{gen}}(\boldsymbol{x}) = G_{\boldsymbol{z}\sim p_z}(\boldsymbol{z})$. С другой стороны, генератор должен научиться обманывать дискриминатор, то есть минимизировать по p_{gen} следующую величину:

$$\mathbb{E}_{\boldsymbol{x}\sim p_{\text{gen}}(\boldsymbol{x})}\left[\log(1 - D(\boldsymbol{x}))\right] = \mathbb{E}_{\boldsymbol{z}\sim p_z(\boldsymbol{z})}\left[\log(1 - D(G(\boldsymbol{z})))\right].$$

Если теперь объединить эти функции в одну, то мы увидим, что фактически дискриминатор и генератор играют между собой в игру, которую в теории игр называют *минимаксной игрой*, решая следующую задачу оптимизации:

$$\min_G \max_D V(D, G), \text{ где}$$

$$V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))].$$

Отсюда и происходит название модели.

Схематически это можно изобразить так, как показано на рис. 8.9. А алгоритм обучения GAN тоже очень простой: мы будем просто поочередно обновлять то веса генератора, то веса дискриминатора, каждый раз считая «противника» фиксированным. Эта идея очень похожа на идею EM-алгоритма, который мы подробно разберем в разделе 10.2. Оказывается, несложно и формально доказать, что идея работает, и при вышеописанном процессе обучения p_{gen} действительно постепенно сходится к p_{data}. Давайте это докажем; далее наше изложение следует [176].

Прежде всего докажем, что для фиксированного генератора G оптимальное распределение дискриминатора D таково:

$$D^*_G(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_{\text{gen}}(\boldsymbol{x})}.$$

И действительно, критерий обучения дискриминатора D при условии некоторого генератора G состоит в том, чтобы максимизировать $V(G, D)$:

$$V(G, D) = \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x}) d\boldsymbol{x} + \int_{\boldsymbol{z}} p_{\boldsymbol{z}}(\boldsymbol{z}) \log(1 - D(g(\boldsymbol{z}))) d\boldsymbol{x} =$$

$$= \int_{\boldsymbol{x}} p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x}) d\boldsymbol{x} + \int_{\boldsymbol{x}} p_{\text{gen}}(\boldsymbol{x}) \log(1 - D(\boldsymbol{x})) d\boldsymbol{x} =$$

$$= \int_{\boldsymbol{x}} \left(p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x}) + p_{\text{gen}}(\boldsymbol{x}) \log(1 - D(\boldsymbol{x})) \right) d\boldsymbol{x}.$$

Но для любых $(a, b) \in \mathbb{R}^2$, кроме нуля, функция $a \log(y) + b \log(1 - y)$ достигает своего максимума на отрезке $[0, 1]$ в точке $\frac{a}{a+b}$ (проверьте это напрямую, дифференцированием!). Поэтому $D^*_G(\boldsymbol{x})$ максимизирует подинтегральное выражение в каждой точке, а значит, и весь интеграл тоже. Отметим, что тогда результат обучения дискриминатора $\max_D V(D, G)$ можно переписать следующим образом:

$$\begin{aligned} C(G) &= \max_D V(D, G) = \\ &= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} \left[\log D^*_G(\boldsymbol{x}) \right] + \mathbb{E}_{z \sim p_z} \left[\log(1 - D^*_G(G(z))) \right] = \\ &= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} \left[\log D^*_G(\boldsymbol{x}) \right] + \mathbb{E}_{\boldsymbol{x} \sim p_g} \left[\log(1 - D^*_G(\boldsymbol{x})) \right] = \\ &= \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_{\text{gen}}(\boldsymbol{x})} \right] + \mathbb{E}_{\boldsymbol{x} \sim p_g} \left[\log \frac{p_g(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_{\text{gen}}(\boldsymbol{x})} \right]. \end{aligned}$$

И теперь можно уже доказывать, что глобальный минимум критерия $C(G)$ достигается тогда и только тогда, когда $p_{\text{gen}} = p_{\text{data}}$. Как мы только что выяснили, при $p_{\text{gen}} = p_{\text{data}}$ мы получим $D^*_G(\boldsymbol{x}) = \frac{1}{2}$. Тогда, подставляя $D^*_G(\boldsymbol{x}) = \frac{1}{2}$ в приведенное выше уравнение для $C(G)$, получим:

$$C(G) = \log \frac{1}{2} + \log \frac{1}{2} = -\log 4.$$

Докажем, что это минимально возможное значение $C(G)$, достижимое только при $p_{\text{gen}} = p_{\text{data}}$. Вычтем выражение $\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}}[-\log 2] + \mathbb{E}_{\boldsymbol{x} \sim p_g}[-\log 2] = -\log 4$ из $C(G) = V(D^*_G, G)$:

$$C(G) = -\log 4 + \text{KL}\left(p_{\text{data}} \middle\| \frac{p_{\text{data}} + p_{\text{gen}}}{2} \right) + \text{KL}\left(p_{\text{gen}} \middle\| \frac{p_{\text{data}} + p_{\text{gen}}}{2} \right),$$

где $\text{KL}(p \| q)$ обозначает расстояние Кульбака — Лейблера между распределениями p и q. Последнее выражение известно в статистике и машинном обучении как *дивергенция Йенсена — Шеннона* (Jensen — Shannon divergence) между моделируемым распределением и порожденным [327]:

$$C(G) = -\log 4 + 2\text{JSD}(p_{\text{data}} || p_{\text{gen}}).$$

Дивергенция Йенсена — Шеннона всегда неотрицательна и равна нулю только при равенстве распределений. А значит, глобальный минимум $C(G)$ достигается при $p_{\text{gen}} = p_{\text{data}}$ и равен $C^* = -\log 4$. Равенство $p_{\text{gen}} = p_{\text{data}}$ означает, что генератор научился идеально воспроизводить данные из обучающей выборки.

Можно доказать и то, что p_{gen} сходится к p_{data} при попеременном обучении генератора и дискриминатора. А именно, если модели, представляющие G и D, достаточно выразительны, и на каждом шаге алгоритма обучения дискриминатор достигает оптимума при условии текущего G, а p_{gen} обновляется так, чтобы улучшать значение критерия

$$\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D^*_G(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_{\text{gen}}(\boldsymbol{x})}[\log(1 - D^*_G(\boldsymbol{x}))],$$

то p_{gen} сходится к p_{data} (формальное доказательство см. в [176]).

Понятно, что на практике D и G представляют собой нейронные сети, и оптимизация происходит по параметрам $\boldsymbol{\theta}_g$, а не по самому распределению p_{gen} в общем виде. Заметим, что для того, чтобы формально следовать доказанному результату, нам нужно было бы после каждого шага обновления генератора полностью дообучать дискриминатор до сходимости, ведь задача классификации, которую он решает, изменилась, и генератору нужно было бы научиться обманывать версию дискриминатора, соответствующую своей текущей версии. Но на практике мы, конечно, не сможем полностью обучать сложную модель на каждом шаге, поэтому мы будем использовать, так сказать, стохастическую версию этого обучения: делать один шаг обучения G, потом несколько шагов обучения D, потом еще один шаг обучения G и т. д.

Несмотря на то, что использование глубоких нейронных сетей приводит к появлению множества критических точек в пространстве параметров, удачные практические результаты их использования позволяют считать нейросети подходящими моделями и (по крайней мере, пока) не обращать внимания на отсутствие теоретических гарантий.

8.4. Практический пример и трюк с логистическим сигмоидом

> В практике должен доказать человек истинность, то есть действительность и мощь, посюсторонность своего мышления. Спор о действительности или недействительности мышления, изолирующегося от практики, есть схоластический вопрос.
>
> *К. Маркс. Тезисы о Фейербахе*

Давайте теперь перейдем к самому интересному — эксперименту. В качестве модельной задачи обучим генератор сэмплировать одномерное нормальное распределение из одномерного равномерного шума. Сначала импортируем TensorFlow и `numpy` и зададим веса для слоев генератора и дискриминатора. Здесь стоит отметить, что задача у нас не слишком сложная, и нет необходимости строить глубокие или «широкие» сети для ее решения. Однако линейным преобразованием из равномерного распределения получить нормальное невозможно, поэтому генератору

необходимо иметь хотя бы два слоя. Так как мы решаем одномерную задачу, внешние слои имеют размерность 1, а для внутреннего слоя хватит и пяти перцептронов. Другой важный и достаточно интуитивно ясный момент состоит в том, что выразительная сила дискриминатора должна быть выше, чем у генератора: если дискриминатор не сможет достичь оптимума, то и штрафы для генератора будут ограничены и не позволят ему обучиться наилучшим образом. Поэтому мы выбрали такую архитектуру сетей:

```
import numpy as np
import tensorflow as tf
gen_weights = dict()
gen_weights['w1'] = tf.Variable(tf.random_normal([1, 5]))
gen_weights['b1'] = tf.Variable(tf.random_normal([5]))
gen_weights['w2'] = tf.Variable(tf.random_normal([5, 1]))
gen_weights['b2'] = tf.Variable(tf.random_normal([1]))

disc_weights = dict()
disc_weights['w1'] = tf.Variable(tf.random_normal([1, 10]))
disc_weights['b1'] = tf.Variable(tf.random_normal([10]))
disc_weights['w2'] = tf.Variable(tf.random_normal([10, 10]))
disc_weights['b2'] = tf.Variable(tf.random_normal([10]))
disc_weights['w3'] = tf.Variable(tf.random_normal([10, 1]))
disc_weights['b3'] = tf.Variable(tf.random_normal([1]))
```

Зададим теперь тензоры, описывающие непосредственно сети. Для начала объявим две «заглушки», одну для априорного шума, другую для выборки из реальных данных (в роли которых у нас выступает нормальное распределение), и зададим тензор, соответствующий $\boldsymbol{x}_g$:

```
z_p = tf.placeholder('float', [None, 1])
x_d = tf.placeholder('float', [None, 1])
g_h = tf.nn.softplus(tf.add(
    tf.matmul(z_p, gen_weights['w1']), gen_weights['b1']))
x_g = tf.add(tf.matmul(g_h, gen_weights['w2']), gen_weights['b2'])
```

Выбор функции активации для скрытого слоя при решении этой задачи не играет большой роли, и вы можете сами поэкспериментировать с другими вариантами и посмотреть, как будут различаться распределения p_{gen} в зависимости от выбора нелинейности. А вот выход генератора мы оставляем линейным, потому что области значений стандартных функций активации так или иначе ограничены, а мы хотим повторить нормальное распределение, которое формально неограничено. Теперь аналогичным образом зададим дискриминатор:

```
def discriminator(x):
    d_h1 = tf.nn.tanh(tf.add(
        tf.matmul(x, disc_weights['w1']), disc_weights['b1']))
    d_h2 = tf.nn.tanh(tf.add(
        tf.matmul(d_h1, disc_weights['w2']), disc_weights['b2']))
```

```
    score = tf.nn.sigmoid(tf.add(
        tf.matmul(d_h2, disc_weights['w3']), disc_weights['b3']))
    return score

x_data_score = discriminator(x_d)
x_gen_score = discriminator(x_g)
```

Поскольку мы хотим использовать одни и те же веса как для оценки выборки из реальных данных, так и для оценки порожденных данных, будем просто вызывать описывающую дискриминатор функцию, передавая ей соответвующий тензор. В выходном слое дискриминатора мы используем сигмоидальную функцию активации, так как дальше собираемся интерпретировать значения как вероятности того, что входные данные были взяты из обучающей выборки, а не порождены генератором.

Функции стоимости для обеих сетей задаются очень просто:

```
D_cost = -tf.reduce_mean(tf.log(x_data_score) + tf.log(1.0 - x_gen_score))
G_cost = tf.reduce_mean(tf.log(1.0 - x_gen_score))
```

Оптимизаторы в TensorFlow по умолчанию решают задачу минимизации, так что `D_cost` мы берем со знаком минус, а `G_cost` — со знаком плюс. Зададим оптимизаторы:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
D_optimizer = optimizer.minimize(D_cost, var_list=disc_weights.values())
G_optimizer = optimizer.minimize(G_cost, var_list=gen_weights.values())
```

Важно отметить, что когда мы оптимизируем дискриминатор, мы не должны обновлять веса генератора и наоборот, поэтому при градиентном спуске мы меняем только соответствующие веса. В последнем отрывке кода мы использовали параметр `learning_rate`, так что пора теперь задать все параметры обучения:

```
batch_size = 64
updates = 40000
learning_rate = 0.01
prior_mu = -2.5
prior_std = 0.5
noise_range = 5.
```

Размер батча при обучении не сильно влияет на итоговый результат, однако задать его все-таки стоит. Для того чтобы продемонстрировать эффективность GAN, мы решили из линейного равномерного шума на отрезке $[-5, 5]$ генерировать нормальное распределение $\mathcal{N}(-2{,}5, 0.5)$, что и записали в параметрах выше. И, раз уж мы заговорили о распределениях, давайте объявим вспомогательные функции:

```
def sample_z(size=batch_size):
    return np.random.uniform(-noise_range, noise_range, size=[size, 1])
def sample_x(size=batch_size, mu=prior_mu, std=prior_std):
    return np.random.normal(mu, std, size=[size, 1])
```

Ну вот мы и добрались до обучения. Создаем сессию TensorFlow и приступаем:

```
init = tf.initialize_global_variables()
sess = tf.Session()
sess.run(init)

for i in range(updates):
    z_batch = sample_z()
    x_batch = sample_x()
    sess.run(D_optimizer, feed_dict={z_p: z_batch, x_d: x_batch})
    z_batch = sample_z()
    sess.run(G_optimizer, feed_dict={z_p: z_batch})
```

В результате обучения такого GAN достаточно быстро можно увидеть, как генератор начинает попадать в априорное распределение. Результаты показаны на рис. 8.10: закрашенная область на графиках — гистограмма сэмплов из настоящего нормального распределения, черная кривая — плотность распределения (тоже эмпирическая), которую в данный момент порождает генератор, а серая кривая — предсказание дискриминатора: значение серой кривой показывает, какую вероятность дискриминатор в этой точке присваивает тому, что данные настоящие.

На рис. 8.10 видно, что этот простой пример GAN ведет себя правильно:

- до начала обучения и генератор, и дискриминатор порождают что-то случайное и никак друг с другом не связаны;
- после трех итераций генератор еще плохо обучился, и дискриминатор его пока «побеждает»: например, пик генератора слева от среднего точно соответствует «провалу» в графике предсказанной дискриминатором вероятности;
- а после ста итераций генератор уже довольно точно соответствует данным, и дискриминатору делать нечего: в зоне, где плотность данных и генератора высока, выход дискриминатора почти точно равен 0,5, то есть он признает свое поражение (а вот слева и справа, видимо, плотность распределения генератора убывает быстрее, чем даже нормальное распределение настоящих данных, так что дискриминатор уверен, что там данные настоящие).

И хотя повторить в точности нормальное распределение не получается, понятно, что на самом деле эту проблему можно решить, увеличивая выразительную силу сети генератора (и дискриминатора тоже — не забывайте, что дискриминатор должен оставаться мощнее).

Пока мы не перешли к более сложным моделям, давайте обсудим функции стоимости. На ранних этапах обучения состязательных моделей дискриминатор достаточно быстро учится отличать примеры из обучающей выборки от порожденных генератором. Поэтому функция стоимости генератора $\log(1 - D(\boldsymbol{x}_g))$ «насыщается» и начинает принимать околонулевые значения, что сильно замедляет обновление весов генератора в ходе обучения. Поэтому вместо этого минимизируют функцию $-\log(D(\boldsymbol{x}_g))$, у которой, очевидно, экстремумы находятся в тех же точках.

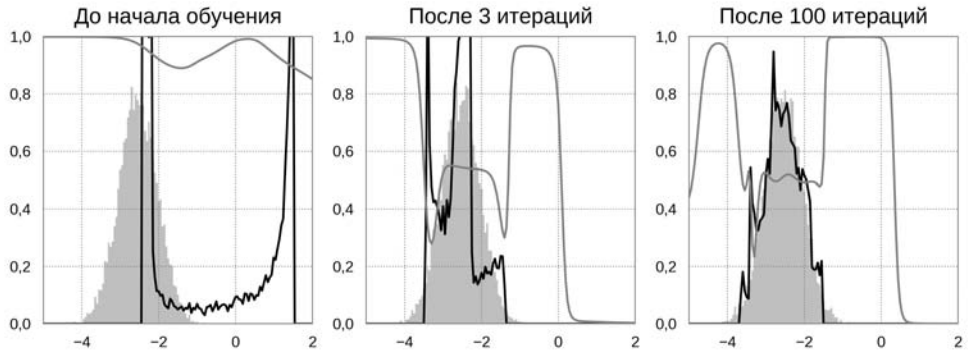


Рис. 8.10. Результат работы GAN

Но это не единственный трюк, который стоит проделать с функциями потерь. Логистическая функция потерь также обладает некоторыми особенностями. При ее вычислении к выходам последнего слоя дискриминатора применяется сигмоидальная функция активации $\sigma(x) = \frac{1}{1+\exp(-x)}$. Мы сталкиваемся с необходимостью вычислять экспоненты, что может привести к ошибкам, так как достаточно относительно небольшого отрицательного значения x, чтобы выйти, например, за максимальное значение типа `float16`.

Поэтому в TensorFlow применяется несложный дополнительный трюк для вычисления логистической функции потерь. Поскольку правильный ответ для примеров из данных равен 1, а для порожденных генератором примеров равен 0, стандартная форма логистической функции потерь распадается на сумму двух логарифмов, $-\log(D(\boldsymbol{x}_d)) - \log(1 - D(\boldsymbol{x}_g))$.

Функцией активации последнего слоя является логистический сигмоид, так что это выражение можно переписать так:

$$-\log(\sigma(\boldsymbol{x}_{\text{logits,data}})) + -\log(1 - \sigma(\boldsymbol{x}_{\text{logits,gen}})),$$

где $\boldsymbol{x}_{\text{logits}}$ — это значения выходного слоя до применения функции активации. Давайте рассмотрим каждое из слагаемых по отдельности (опуская для простоты в каждом случае нижний индекс при x):

$$-\log(\sigma(x)) = -\log\left(\frac{1}{1+\exp(-x)}\right) = \log(1+\exp(-x)) =$$
$$= \log\left(1+\frac{1}{\exp(x)}\right) = \log\left(\frac{1+\exp(x)}{\exp(x)}\right) = \log(1+\exp(x)) - x,$$

то есть $\log(1+\exp(-x)) = \log(1+\exp(x)) - x$.

Очевидно, что если для положительных значений x мы будем вычислять функцию с левой стороны тождества, а для отрицательных — с правой, то проблема переполнения отпадет. Но можно ли записать эти два выражения в одной и той же форме так, чтобы в показателе экспоненты всегда стояло отрицательное число? Оказывается, можно! И вот как это выглядит:

$$\max(x, 0) - x + \log(1 + \exp(-|x|)).$$

Тогда, если x положительный, то $\max(x, 0)$ взаимоуничтожается с x, а $-|x|$ равно $-x$, и остается левая часть тождества, а если x отрицательный, то $\max(x, 0) = 0$, $-|x| = x$, и перед нами правая сторона тождества. Осталось только заметить, что $\max(x, 0)$ — это как раз и есть наша любимая функция активации ReLU, и окончательный вариант этой части суммы выглядит в TensorFlow так:

```
tf.nn.relu(x) - x + tf.log(1.0 + tf.exp(-tf.abs(x)))
```

Со вторым слагаемым все аналогично:

$$-\log(1-\sigma(x)) = -\log\left(1 - \frac{1}{1+\exp(-x)}\right) = -\log\left(\frac{\exp(-x)}{1+\exp(-x)}\right) = \\ = \log\left(\frac{1+\exp(-x)}{\exp(-x)}\right) = \log(1+\exp(x)),$$

а с другой стороны:

$$\log\left(\frac{1+\exp(-x)}{\exp(-x)}\right) = \log(1+\exp(-x)) + x,$$

откуда легко получить аналогичную форму:

$$\max(x, 0) + \log(1 + \exp(-|x|)).$$

С учетом вышесказанного нам остается совсем немного модифицировать код, а именно сначала изменить функцию `discriminator(x)`:

```
def discriminator(x):
    d_h1 = tf.nn.tanh(tf.add(
        tf.matmul(x, disc_weights['w1']), disc_weights['b1']))
    d_h2 = tf.nn.tanh(tf.add(
        tf.matmul(d_h1, disc_weights['w2']), disc_weights['b2']))
    logits = tf.add(tf.matmul(d_h2, disc_weights['w3']), disc_weights['b3'])
    return logits
```

А затем переопределить функции стоимости:

```
D_plus_cost = tf.reduce_mean(tf.nn.relu(x_data_score) - x_data_score +
    tf.log(1.0 + tf.exp(-tf.abs(x_data_score))))
```

```
D_minus_cost = tf.reduce_mean(tf.nn.relu(x_gen_score) +
    tf.log(1.0 + tf.exp(-tf.abs(x_gen_score))))
G_cost = tf.reduce_mean(tf.nn.relu(x_gen_score) - x_gen_score +
    tf.log(1.0 + tf.exp(-tf.abs(x_gen_score))))
D_cost = D_plus_cost + D_minus_cost
```

Теперь мы снова можем убедиться в том, что все работает (проверьте сами!), и пора переходить к более интересным задачам. В следующем разделе мы дадим краткий обзор интересных современных применений GAN, покажем структуру состязательного автокодировщика, а затем на практическом примере увидим, как он работает.

8.5. Архитектуры, основанные на GAN

> Строительство на вражде — строительство на вулкане. Взрыв — и снова царство смерти и разрушения.
>
> *Послание Патриарха Тихона*
> *чадам Православной Российской Церкви, 1919 г.*

Прежде чем переходить к современным архитектурам порождающих состязательных сетей, начнем с нескольких замечаний. Во-первых, вообще говоря, мы пока не очень хорошо понимаем, как GAN работает и почему он делает это так хорошо (а GAN'ы действительно хороши!).

Одно возможное объяснение [185] заключается в том, что задача минимаксоптимизации, о которой мы говорили в разделе 8.3, содержит оптимизацию не расстояния Кульбака — Лейблера $\mathrm{KL}(p_{\text{data}}\|p_{\text{gen}})$, а дивергенции Йенсена — Шеннона, что больше похоже на $\mathrm{KL}(p_{\text{gen}}\|p_{\text{data}})$. Разница между ними проиллюстрирована на рис. 8.11, где мы сэмплировали точки из p_{data} как смеси двух нормальных распределений, а из p_{gen} — как из распределения, минимизирующего то или иное расстояние Кульбака — Лейблера:

- $\mathrm{KL}(p_{\text{data}}\|p_{\text{gen}})$ минимизируется там, где все большие значения p_{data} имеют разумные вероятности по распределению p_{gen}; это значит, что если p_{data} имеет много ярко выраженных пиков (как почти всегда и бывает в сложных задачах машинного обучения), p_{gen} будет пытаться «накрыть» их все, «размазывая» результат (см. рис. 8.11, *а*);
- $\mathrm{KL}(p_{\text{gen}}\|p_{\text{data}})$ минимизируется там, где все большие значения p_{gen} имеют разумные вероятности по распределению p_{data}, то есть в таком мультимодальном случае p_{gen} просто «выберет» один из пиков p_{data} (см. рис. 8.11, *б*).

Понятно, что с точки зрения порождения новых примеров второй вариант гораздо лучше первого: возможно, мы не научимся порождать все породы котиков на свете, но зато и не будем порождать «усредненных котиков», не имеющих никакого отношения к реальности (как между пиками p на рис. 8.11).

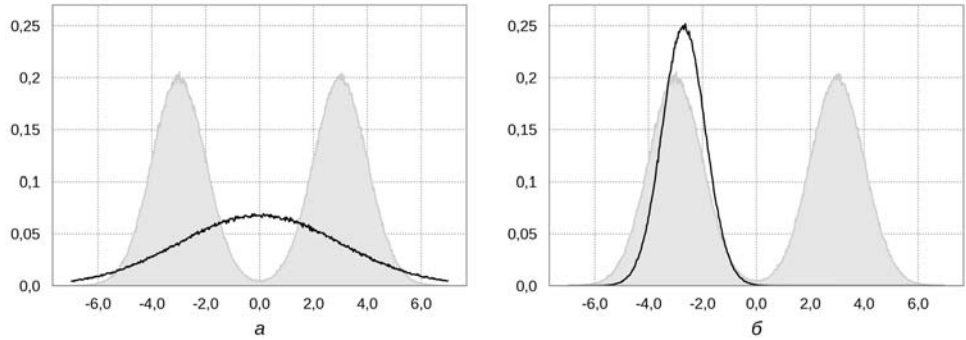


Рис. 8.11. Два разных KL-расстояния между p_{gen} (черная кривая) и p_{data} (серая область): *а* — минимизируем $\text{KL}(p_{\text{data}} \| p_{\text{gen}})$; *б* — минимизируем $\text{KL}(p_{\text{gen}} \| p_{\text{data}})$

Но нет, оказывается, что можно построить GAN и с другими функциями ошибки, например, в [397] построена конструкция так называемых *f-GAN*, которые могут оптимизировать любую функцию из класса f-дивергенций, к которым относится и дивергенция Кульбака — Лейблера, и Йенсена — Шеннона, и многие другие. И все равно GAN работает хорошо, и сэмплы из него получаются достаточно четкие.

Во-вторых, из этого обсуждения отчасти следует и одна из главных на данный момент проблем GAN: схлопывание мод распределения (mode collapse). Она отлично проиллюстрирована на рис. 8.11, *б*: оптимальное распределение q выбирает одну моду (локальный максимум) распределения p, а второй оказывается совершенно не покрыт, и в результате мы порождаем одну конкретную породу котиков, а не любую возможную. Полное решение этой проблемы еще впереди.

В-третьих, мы уже упоминали, что GAN'ы обучать сложно, и модели это хрупкие, так что стоит иметь в виду ряд практических советов [176, 245]:

- метки классов: если данные делятся на k классов (например, кошечки и собачки), можно определить дискриминатор как классификатор на $k+1$ класс: k классов, которые выделяются в данных, и еще один класс, соответствующий сэмплам из генератора; практика показывает, что сэмплы становятся гораздо лучше, если подавать на вход метки классов, причем даже в том случае, если мы даем их только дискриминатору [245];
- дискриминация по мини-батчам (mini-batch discrimination): одна из эвристик, помогающих справиться с проблемой схлопывания и сделать генератор более разнообразным, состоит в том, чтобы подавать дискриминатору сэмплы целыми мини-батчами и подсчитывать некоторую метрику схожести между сэмплами в мини-батче в качестве вспомогательного входа для дискриминатора; это часто приводит к существенным улучшениям в генерации [176];

- сглаживание меток (label smoothing) для дискриминатора: он в конструкции GAN оценивает отношение распределений, а нейронные сети часто любят быть слишком уверены, выдавать значения слишком близкие к нулю или единице; чтобы исправить это, целевое значение дискриминатора для точек из данных часто меняют с 1 на, к примеру, 0,9, а на сгенерированных точках оставляют как есть; иначе говоря, мы обучаем дискриминатор так, чтобы в идеале на точках из данных он был на 90 % уверен, что это точки из данных, а не на 100 %; (см. также [448], где аналогичный трюк оказывается полезен для сверточных сетей и распознавания изображений);
- виртуальная нормализация по мини-батчам: обычная нормализация очень полезна, и в генератор ее добавлять очень хотелось бы, но в результате получается так, что порожденные сэмплы в рамках одного мини-батча получаются сильно скоррелированными (у них ведь общие параметры batchnorm-слоя); чтобы этого избежать, можно использовать в генераторе один и тот же выбранный из данных виртуальный «эталонный батч» (reference batch) для каждого примера вместо того, чтобы считать среднее и ковариации отдельно;
- дисбаланс между генератором G и дискриминатором D: интуитивно может показаться, что нужно поддерживать какой-то «баланс» между G и D, не давая никому вырываться вперед, но на самом деле, как мы уже говорили, функция ошибки для G имеет куда больше смысла, когда D оптимален, то есть лучше, если D будет «сильнее»; на практике это значит, что D может быть более сложной сетью, более выразительной.

В целом, теория порождающих состязательных сетей — это сейчас один из фронтиров обучения глубоких сетей; нас наверняка ждет много интересных новых результатов в этом направлении. А теперь давайте посмотрим, какие конструкции GAN'ов используются на практике и что с ними можно сделать...

Первые яркие применения порождающих состязательных сетей были связаны с архитектурой DCGAN (Deep Convolutional Generative Adversarial Networks) [432]. Это обычный GAN, в котором генератор и дискриминатор представляют собой глубокие сверточные сети, что логично для обработки изображений. Однако есть несколько важных особенностей, которые стоит иметь в виду, если вы сами захотите применить GAN к тем или иным картинкам:

- вместо субдискретизации используются дополнительные сверточные уровни, в которых свертки участвуют с пробелами (strided convolutions, как на рис. 5.2); эта идея известна как «полностью сверточные» сети (all convolutional nets) [511];
- полносвязные скрытые слои тоже не используются, примерно как в [376]; фактически от сверточной сети остаются только сами свертки;
- а вот нормализация по мини-батчам используется как в генераторе, так и в дискриминаторе;
- в генераторе везде используется ReLU, а в дискриминаторе — протекающий ReLU (Leaky ReLU, см. раздел 3.3).

Результаты DCGAN даже в исходной статье очень интересны[1]. Модель была обучена, в частности, на датасете интерьеров спален LSUN (Large-Scale Scene Understanding Challenge) [337] и на собранном авторами датасете из трех миллионов человеческих лиц. В результате не просто получился генератор, который производит разумные интерьеры и человеческие лица, но и у пространства скрытых факторов нашлись очень интересные свойства. Например, оказалось, что в пространстве скрытых факторов иногда работает примерно такая же векторная арифметика, которую мы видели в распределенных представлениях слов в разделе 7.2: можно, например, вычесть из фотографий мужчин в очках фотографии мужчин без очков, прибавить женщин без очков и получить женщин в очках! А «прогулки» по пространству скрытых факторов показывают, что по мере продвижения от одного интерьера к другому модель все время порождает вполне разумные изображения интерьеров, не пытаясь усреднять и размазывать фотографии, а двигаясь в некоторой «логической» последовательности.

Есть много ярких и забавных примеров применения DCGAN: например, в [175] им генерируют пиксельную графику для восьмибитных игр. А в [69] DCGAN-подобной архитектурой порождают «картины» в стиле абстрактного экспрессионизма; эксперименты на людях показали, что порожденные GAN'ом картины вполне на уровне человеческих по таким категориям, как «новизна», «сложность», «преднамеренность» (intentionality), да и просто по ответу на вопрос «человек ли это рисовал» (впрочем, слишком серьезно работу [69] принимать пока не нужно, опрашивали там буквально человек двадцать).

Уже появились и практически важные задачи, которые такие архитектуры решают лучше всех. Например, в [421] решается задача *повышения разрешения* изображения (image super-resolution): можем ли мы разумным образом увеличить разрешение фотографии, добавить деталей так, чтобы они казались правдоподобными на человеческий взгляд (понятно, что речь не идет о том, чтобы восстановить «настоящие» детали, такой информации в фотографии низкого качества просто нет). Архитектура SRGAN из [421] очень похожа на DCGAN, и результаты получаются очень хорошие. Похожа на это и задача восстановления МРТ-изображений в медицине (MRI reconstruction), которую решают с помощью GAN в [106].

Следующее очень естественное расширение архитектуры состязательных сетей — *условный GAN* (conditional GAN) [366]. Эта модель использует дополнительную информацию $\boldsymbol{y}$, которая часто доступна вместе с примерами $\boldsymbol{x}$; например, это может быть метка класса изображения (кошка или собака, номер цифры в MNIST и так далее). Мы подаем $\boldsymbol{y}$ вместе со случайным вектором на вход генератору, а также будем подавать тот же $\boldsymbol{y}$ на вход дискриминатора для различения (см. рис. 8.12, *а*); в результате генератор пытается строить модель условного распределения $p_{\text{gen}}(\boldsymbol{x} \mid \boldsymbol{y}) = G(\boldsymbol{z}, \boldsymbol{y})$, а дискриминатор $D(\boldsymbol{x} \mid \boldsymbol{y})$ строит распределение вероятности того, что вход «настоящий», тоже при условии $\boldsymbol{y}$.

[1] Как и в разделе 8.1, красивых картинок, порожденных DCGAN, мы здесь приводить не будем — смотрите по ссылкам, а в книге мы будем говорить скорее об архитектурах.

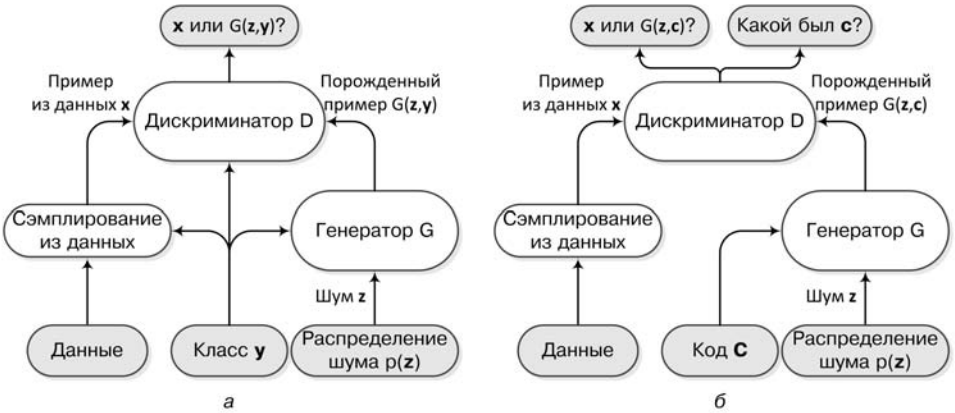


Рис. 8.12. Архитектуры модификаций GAN: *а* — условный GAN [366]; *б* — InfoGAN [248]

В исходной статье [366] применения условных GAN были не слишком впечатляющими, но быстро стали появляться очень интересные новые работы[1]. Например, в [6] в качестве объектов используются фотографии человеческих лиц, а в качестве дополнительной информации — возраст человека на фотографии. Такой условный GAN способен порождать лица заданного возраста; более того, по фотографии лица можно сначала отобразить его в пространство признаков, а потом изменить признак возраста и породить новую фотографию, таким образом «состарив» или «омолодив» человека на фотографии! Для этого нужна чуть более сложная архитектура, но вдаваться в подробности мы сейчас не будем. Другое развитие идея изменения человеческих лиц получила в [251], где условия имеют физический смысл — «светлые волосы», «лысина», «очки», — и в модели есть специальный кодировщик, который распознает эти признаки. Получается, что теоретически можно распознать признаки на фотографии, искусственно заменить некоторые из них («перекрасить волосы», «убрать лысину», «надеть очки»), а затем породить фотографию с другими признаками. Конечно, пока это работает далеко не идеально, и мы еще не живем в киберпанк-антиутопии, но прогресс не стоит на месте.

Еще одна интересная модификация условных GAN, *InfoGAN* [248], добавляет к обычной задаче оптимизации, решаемой в GAN, дополнительные теоретико-информационные ограничения. Задача состоит в том, чтобы обучить на скрытом слое не просто «хорошее» представление, из которого можно сэмплировать, а «распутанное» представление (disentangled representation), в котором отдельные признаки имеют естественную интерпретацию. Например, хотелось бы, чтобы признаки у GAN, генерирующей человеческое лицо, соответствовали цвету глаз, форме

[1] Мы, конечно, не сможем их все перечислить, да и новые интересные GAN'ы появляются буквально каждую неделю; см., например, список ссылок в [217].

носа и тому подобным естественным особенностям. Авторы [248] добиваются этого несложным, но концептуальным способом:

- явным образом разделим вектор шума на две части, «настоящий» несжимаемый шум $\boldsymbol{z}$ и структурированный шум, или «код» $\boldsymbol{c} = c_1, c_2, \ldots, c_L$;
- сделаем предположение о том, что распределение кода полностью факторизуется, $p(\boldsymbol{c}) = p(c_1)p(c_2)\ldots p(c_L)$; теперь генератор зависит от обеих частей, $G(\boldsymbol{z},\boldsymbol{c})$, а это предположение говорит, что признаки в $\boldsymbol{c}$ содержательные и независимые;
- но при этом генератор пока что не обязан вообще учитывать $\boldsymbol{c}$; чтобы его заставить, добавим в функцию ошибки максимизацию взаимной информации $I(\boldsymbol{c}; G(\boldsymbol{z},\boldsymbol{c}))$ между кодом $\boldsymbol{c}$ и выходом генератора; это значит, что генератор будет пытаться как можно сильнее учитывать $\boldsymbol{c}$ в своем выходе, максимизировать влияние $\boldsymbol{c}$ на $G(\boldsymbol{z},\boldsymbol{c})$, то есть по сути будет пытаться сделать так, чтобы по $G(\boldsymbol{z},\boldsymbol{c})$ можно было восстановить $\boldsymbol{c}$ (см. рис. 8.12, *б*).

В результате структура модели и обучение усложняются не сильно — по сути, добавляется один полносвязный слой, который считает и максимизирует нижнюю оценку $I(\boldsymbol{c}; G(\boldsymbol{z},\boldsymbol{c}))$. Но скрытые представления получаются действительно «распутанными»: например, на MNIST выделяются признаки, соответствующие самой цифре, ее ширине, углу поворота и т. д.

Другое естественное применение условий в условном GAN — это более детальное указание на то, что именно и как нужно нарисовать. Например, в [311, 508] решают поражающую воображение задачу: сгенерировать фотореалистичный снимок по заданному *текстовому описанию*! При этом текст сжимается в распределенное представление с помощью рекуррентного кодировщика, а затем эти признаки используются как условие в GAN. В [311] к текстовому описанию добавляются еще и указания в координатах, где должны быть те или иные части изображения, а в [508] работают уже напрямую из текста. В обеих работах модели обучаются на наборе фотографий птиц с описаниями, так что в результате можно подать на вход описание вроде «Маленькая желтая птичка с черной короной и коротким черным острым клювом», и на выходе действительно получится довольно похожая на это описание «фотография». StackGAN — это хороший пример современной достаточно сложной архитектуры нейронной сети, главным образом основанной на идее состязательных сетей, но «впитавшей» при этом в себя много разных конструкций:

- сначала текст превращается в векторное представление с помощью рекуррентного кодировщика;
- к полученным признакам добавляются дополнительные случайные компоненты, и результат служит входом для GAN;
- но GAN в модели сразу два (поэтому она и называется StackGAN): сначала по текстовому описанию генерируется маленький эскиз размером 64×64 пиксела, а потом уже из него получается картинка 256×256, то есть фактически второй уровень решает задачу повышения разрешения.

С проблемой схлопывания мод распределения весьма успешно борется работа [545], в которой предлагается модификация алгоритма обучения обычного GAN. Идея состоит в том, чтобы «заглянуть в будущее» и превратить обучение генератора в GAN в своеобразный рекуррентный процесс, предсказывая реакцию дискриминатора на обучение генератора прямо в ходе этого обучения. Обычно генератор оптимизирует $V(D,G) = V(\boldsymbol{\theta}_D, \boldsymbol{\theta}_G)$, где D — текущий дискриминатор, $\boldsymbol{\theta}_D$ — его параметры, а $\boldsymbol{\theta}_G$ — параметры генератора. И мы знаем, что собираемся оптимизировать D градиентным спуском, точнее подъёмом по функции V:

$$\boldsymbol{\theta}_D^* = \lim_{k\to\infty} \boldsymbol{\theta}_D^{(k)}, \quad \text{где} \quad \boldsymbol{\theta}_D^{(0)} = \boldsymbol{\theta}_D, \quad \boldsymbol{\theta}_D^{(k+1)} = \boldsymbol{\theta}_D^{(k)} + \eta^{(k)} \frac{\partial V(\boldsymbol{\theta}_G, \boldsymbol{\theta}_D^{(k)})}{\partial \boldsymbol{\theta}_D^{(k)}}.$$

Ключевая идея [545] состоит в том, чтобы посмотреть на все это выражение целиком как на единую формулу от $\boldsymbol{\theta}_G$ и заметить, что мы можем тем самым напрямую оптимизировать $V(\boldsymbol{\theta}_D^*, \boldsymbol{\theta}_G)$ по переменным $\boldsymbol{\theta}_G$. Конечно, буквально $\boldsymbol{\theta}_D^*$, результат бесконечного итеративного процесса, использовать не получится, но заглянуть в будущее на несколько итераций и взять, скажем, $\boldsymbol{\theta}_D^{(5)}$ вполне возможно.

Получается, что генератор соперничает не с текущим дискриминатором, а с идеализированным, который уже немножко обучился распознавать текущий генератор; поэтому генератору приходится обманывать куда более хитрого противника, чем в обычном GAN. В частности, это значит, что если генератору захочется все время выбирать одну моду из мультимодального распределения, такой идеализированный дискриминатор быстро, за те же пять итераций, сможет обучиться штрафовать эту моду, чем заранее отобьет охоту генератору обучаться в этом унимодальном направлении. Обратной стороной этой идеи является высокая вычислительная сложность: функция ошибки со встроенными пятью итерациями обучения дискриминатора действительно становится гораздо сложнее.

Последняя важная модификация основной конструкции порождающих состязательных сетей, которую мы рассмотрим в этой главе, — *состязательные автокодировщики* (adversarial autoencoders, AAE) [4].

Структура AAE показана на рис. 8.13. Основная идея здесь состоит в том, чтобы превратить обычный автокодировщик, о котором мы говорили в разделе 5.5, в порождающую модель. Как мы уже обсуждали в разделе 8.2, автокодировщик может развернуть скрытое представление в «настоящий» объект, но это не помогает порождению напрямую: подходящие комбинации скрытых факторов взять неоткуда, а если порождать их случайно, ничего разумного не получится.

Соперничающий автокодировщик пытается решить как раз последнюю проблему. Как показано на рис. 8.13, дискриминатор в AAE пытается отличить распределение скрытых факторов, полученное кодировщиком, от какого-нибудь заданного, фиксированного распределения, например стандартного нормального распределения $\mathcal{N}(0,1)$ по каждому скрытому фактору.

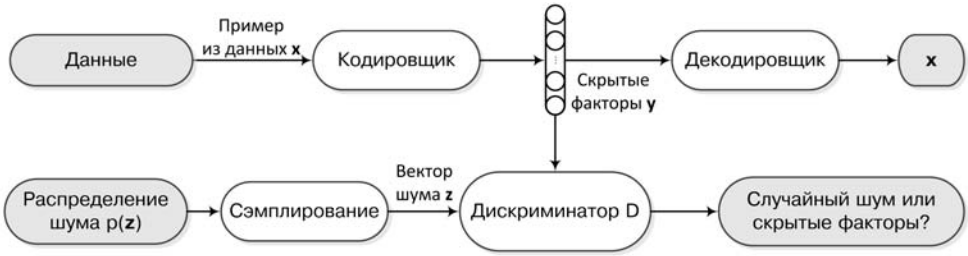


Рис. 8.13. Архитектура состязательного автокодировщика [4]

В результате должно получиться так, что распределение скрытых признаков, соответствующее распределению входных объектов, неотличимо от заранее заданного распределения... а это значит, что мы можем просто взять новый набор скрытых признаков по этому известному распределению, развернуть его декодирующей частью автокодировщика и получить новый сэмпл нужного объекта.

Такую конструкцию можно без особых проблем модифицировать и в *условный AAE*. Дискретное условие, например метка класса, в дискриминаторе будет выглядеть просто: он будет пытаться отличить каждый класс от своего собственного распределения скрытых факторов. В результате AAE может просто «разнести» разные классы объектов в разные части пространства скрытых факторов (насколько это возможно, конечно же; иногда объекты действительно похожи на несколько классов сразу, и тут ничего не поделаешь).

Давайте детально разберем пример состязательного автокодировщика, обучающегося порождать рукописные цифры из датасета MNIST. Это будет, пожалуй, самый сложный пример кода в этой книге, но в нем мы наглядно увидим все составные части порождающей состязательной сети. Сначала импортируем нужные объекты, загрузим MNIST и определим вспомогательные функции:

```
import numpy as np, tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

def batch_gen(data, batch_n):
    inds = range(data.shape[0])
    np.random.shuffle(inds)
    for i in range(data.shape[0] / batch_n):
        ii = inds[i*batch_n:(i+1)*batch_n]
        yield data[ii, :]

def he_initializer(size):
    return tf.random_normal_initializer(mean=0.0,
        stddev=np.sqrt(1./size), seed=None, dtype=tf.float32)
```

```
def linear_layer(tensor, input_size, out_size, init_fn=he_initializer,):
    W = tf.get_variable('W', shape=[input_size, out_size],
        initializer=init_fn(input_size))
    b = tf.get_variable('b', shape=[out_size],
        initializer=tf.constant_initializer(0.1))
    return tf.add(tf.matmul(tensor, W), b)

def sample_prior(loc=0., scale=1., size=(64, 10)):
    return np.tanh(np.random.normal(loc=loc, scale=scale, size=size))
```

Здесь `batch_gen` последовательно выдает мини-батчи данных в случайном порядке, `he_initializer` задает инициализацию весов для ReLU-нейронов по Xe (см. раздел 4.2), `linear_layer` — это вспомогательная функция, которая задает стандартный линейный слой с матрицей весов, инициализирующейся `init_fn`, и вектором свободных членов, инициализирующимся константами, а `sample_prior` выдает векторы случайных нормально распределенных чисел, пропущенные через гиперболический тангенс; эта функция пригодится для порождения мини-батчей случайных векторов для генератора.

Раз уж это все равно самый сложный пример кода в книге, мы оформим его так, как обычно оформляются модели глубоких нейронных сетей на практике: в виде класса, поля которого задают параметры и переменные модели, методы делают собственно обучение, а при инициализации объекта класса создается модель.

Давайте с инициализации и начнем. Нижеследующий большой инициализатор конструирует всю сеть, определяет все функции ошибки и создает оптимизаторы для обучения. Для сокращения метода мы используем пока еще не определенные методы `self._encoder`, `self._decoder` и `self._discriminator`, которые будут задавать структуру собственно сетей. А больше всего нас будут интересовать функции ошибки; мы их по мере появления прокомментируем.

```
class AAE(object):
    def __init__(self, batch_size=64, input_space=28*28,
                 latent_space=10, p=3., middle_layers=None,
                 activation_fn=tf.nn.tanh, learning_rate=0.001, l2_lambda = 0.001,
                 initializer_fn=he_initializer):

        self.batch_size = batch_size
        self.input_space = input_space
        self.latent_space = latent_space
        self.p = p
        self.middle_layers = [1024, 1024]
        self.activation_fn = activation_fn
        self.learning_rate = learning_rate
        self.initializer_fn = initializer_fn

        tf.reset_default_graph()
```

```
        self.input_x = tf.placeholder(tf.float32, [None, input_space])
        self.z_tensor = tf.placeholder(tf.float32, [None, latent_space])

        with tf.variable_scope("encoder"):
            self._encoder()
        self.encoded = self.encoder_layers[-1]

        with tf.variable_scope("decoder"):
            self.decoder_layers = self._decoder(self.encoded)
            self.decoded = self.decoder_layers[-1]
            tf.get_variable_scope().reuse_variables()
            self.generator_layers = self._decoder(self.z_tensor)
            self.generated = tf.nn.sigmoid(
                self.generator_layers[-1], name="generated")

        sizes = [64, 64, 1]
        with tf.variable_scope("discriminator"):
            self.disc_layers_neg = self._discriminator(self.encoded, sizes)
            self.disc_neg = self.disc_layers_neg[-1]
            tf.get_variable_scope().reuse_variables()
            self.disc_layers_pos = self._discriminator(self.z_tensor, sizes)
            self.disc_pos = self.disc_layers_pos[-1]
```

На этом месте у нас готовы выходы всех сетей. Ошибка дискриминатора `disc_loss` будет складываться из положительной и отрицательной ошибки, с трюком, описанным в разделе 8.4. Ошибку кодировщика `enc_loss` мы уже обсуждали в том же разделе. А новой для нас будет ошибка автокодировщика `ae_loss`, которая показывает, насколько успешно декодировщик восстанавливает вход. Кроме того, мы добавим L_2-регуляризатор `l2_loss` на все веса сети.

```
self.pos_loss = tf.nn.relu(self.disc_pos) - self.disc_pos
              + tf.log(1.0 + tf.exp(-tf.abs(self.disc_pos)))
self.neg_loss = tf.nn.relu(self.disc_neg)
              + tf.log(1.0 + tf.exp(-tf.abs(self.disc_neg)))
self.disc_loss = tf.reduce_mean(tf.add(self.pos_loss, self.neg_loss))
self.enc_loss = tf.reduce_mean(tf.subtract(self.neg_loss, self.disc_neg))
batch_logloss = tf.reduce_sum(tf.nn.sigmoid_cross_entropy_with_logits(
              logits=self.decoded, labels=self.input_x), 1)
self.ae_loss = tf.reduce_mean(batch_logloss)
disc_ws = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='discriminator')
ae_ws = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='encoder') +
       tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope='decoder')
self.l2_loss = tf.multiply(tf.reduce_sum([tf.nn.l2_loss(ws) for ws in ae_ws]),
                           l2_lambda)
```

Теперь ошибка генератора `gen_loss` складывается из `enc_loss`, `ae_loss` и `l2_loss`, и мы создаем два оптимизатора, для `gen_loss` и для `disc_loss`. Остаток инициализатора состоит из стандартного запуска сессии TensorFlow.

```
self.gen_loss = tf.add(tf.add(self.enc_loss, self.ae_loss), self.l2_loss)
with tf.variable_scope('optimizers'):
    self.train_discriminator = tf.train.RMSPropOptimizer(self.learning_rate)
                                    .minimize(self.disc_loss, var_list=disc_ws)
    self.train_generator = tf.train.RMSPropOptimizer(self.learning_rate)
                                    .minimize(self.gen_loss, var_list=ae_ws)
self.sess = tf.Session()
init = tf.global_variables_initializer()
self.sess.run(init)
```

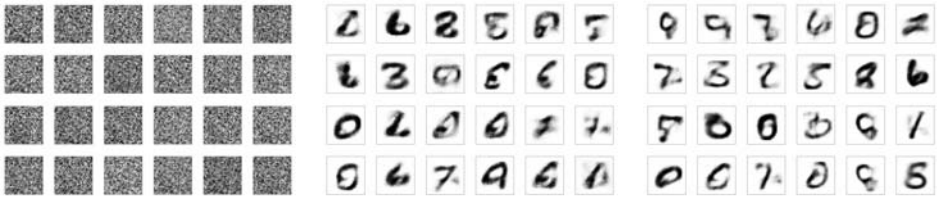
Определим теперь структуру трех составных частей AAE.

```
def _encoder(self):
    sizes = [self.input_space] + self.middle_layers + [self.latent_space]
    self.encoder_layers = [self.input_x]
    for i in range(len(sizes) - 1):
        with tf.variable_scope('layer-%s' % i):
            linear = linear_layer(self.encoder_layers[-1], sizes[i], sizes[i+1])
            self.encoder_layers.append(self.activation_fn(linear))

def _decoder(self, tensor):
    sizes = [self.latent_space] + self.middle_layers[::-1]
    decoder_layers = [tensor]
    for i in range(len(sizes) - 1):
        with tf.variable_scope('layer-%s' % i):
            linear = linear_layer(decoder_layers[-1], sizes[i], sizes[i+1])
            decoder_layers.append(self.activation_fn(linear))
    with tf.variable_scope('output-layer'):
        linear = linear_layer(decoder_layers[-1], sizes[-1], self.input_space)
        decoder_layers.append(linear)
    return decoder_layers

def _discriminator(self, tensor, sizes):
    sizes = [self.latent_space] + sizes + [1]
    disc_layers = [tensor]
    for i in range(len(sizes) - 1):
        with tf.variable_scope('layer-%s' % i):
            linear = linear_layer(disc_layers[-1], sizes[i], sizes[i+1])
            disc_layers.append(self.activation_fn(linear))
    with tf.variable_scope('class-layer'):
        linear = linear_layer(disc_layers[-1], sizes[-1], self.input_space)
        disc_layers.append(linear)
    return disc_layers
```

И можно создавать метод для обучения. Обратите внимание, что то, какой из двух оптимизаторов запускать, определяется текущими значениями ошибок: пока ошибка кодировщика больше заданного значения, мы дообучаем генератор, а когда становится меньше, возвращаемся к дискриминатору.



а *б* *в*

Рис. 8.14. Результаты сэмплирования из состязательного автокодировщика, обученного на MNIST: *а* — после первой эпохи обучения; *б* — после 10 эпох; *в* — после 20 эпох

```
def train(self):
    sess = self.sess
    test_x = mnist.test.images
    gloss = 0.69

    for i in range(1000):
        batch_x, _ = mnist.train.next_batch(self.batch_size)
        if gloss > np.log(self.p):
            gloss, _ = sess.run([self.enc_loss, self.train_generator],
                                feed_dict={self.input_x: batch_x})
        else:
            batch_z = sample_prior(
                scale=1.0, size=(len(batch_x), self.latent_space))
            gloss, _ = sess.run([self.enc_loss, self.train_discriminator],
                        feed_dict={self.input_x: batch_x, self.z_tensor: batch_z})
        if i % 100 == 0:
            gtd = aae.sess.run(aae.generated,
                        feed_dict={aae.z_tensor: sample_prior(size=(4, 10))})
            plot_mnist(gtd.reshape([4, 28, 28]), [1, 4])
```

И теперь можно запускать обучение, а там и посмотреть, что получится.

```
aae = AAE()
aae.train()
```

Результаты сэмплирования на нескольких разных этапах обучения показаны на рис. 8.14; как видите, даже с такой простой структурой кодировщика, декодировщика и дискриминатора у нас быстро, уже после 10–20 эпох обучения, начинают получаться довольно разумные сэмплы.

Дальнейшее развитие состязательных автокодировщиков продолжается прямо сейчас. Например, в [100] предлагаются конструкции шумоподавляющих AAE (в обычной конструкции не вполне ясно, где и как правильно вносить шум, здесь могут быть варианты). А работа [345] развивает идею PixelCNN, о которой мы

говорили в разделе 8.2. В PixelCNN уже можно было построить условное распределение при условии того или иного класса (например, класса из ImageNet) и порождать пиксел за пикселом изображение «на заданную тему». А в модели PixelGAN [345] к PixelCNN добавляется дискриминатор, который, как и в обычном состязательном автокодировщике, приводит распределение скрытых признаков к заданному, что позволяет отделить стиль изображения от его содержания (например, класс цифры в MNIST от почерка) и делает условное порождение еще лучше.

И еще один пример, в котором мы просто никак не можем себе отказать. В работах [97, 130] авторы этой книги использовали состязательный автокодировщик для порождения молекул, которые могли бы быть хорошими кандидатами для лекарственных средств. В химии, молекулярной биологии и фармацевтике сложные структуры (например, сложные молекулы) иногда представляются в виде так называемых «отпечатков» (fingerprints), фактически просто скрытых факторов в виде векторов битов или чисел. Поэтому архитектура условного AAE для этой задачи выглядела так: на вход подаются отпечатки молекул и их концентрации, а дискриминатор, как водится, пытался отличить стандартное нормальное распределение от распределения признаков на скрытом слое. Но скрытый слой при этом был искусственно разделен на две части: «обычные признаки» и специальный признак, который должен быть показывать, насколько эффективна данная молекула против условной болезни. Таким образом, для генерации можно взять обычное нормальное распределение на «обычных признаках» и желаемую (то есть, скорее всего, высокую) эффективность на этом специальном признаке... и, по идее, должна получаться молекула, которая эффективно борется с нужной болезнью. Все эти исследования по сути еще только начинаются, но первые результаты уже получаются очень многообещающими.

Мы уже говорили о конструкциях, которые порождают картинку по текстовому описанию. Но почему же практически все примеры, где мы что-то порождали состязательными сетями, касались исключительно изображений? Где компьютерные писатели и поэты, почему мы не говорили о GAN'ах в главе 7? Здесь появляется очень интересное направление для будущих исследований. Дело в том, что конструкция состязательных сетей хорошо приспособлена для того, чтобы порождать *непрерывные* объекты (например, векторы из чисел, которыми по сути являются изображения). А с порождением *дискретных* объектов возникает проблема: небольшое изменение в сети не приводит к изменениям, и функция ошибки GAN становится кусочно-постоянной, что не позволяет запускать по ней градиентный спуск. Переход к распределенным представлениям слов ситуацию не спасет: небольшое изменение в пространстве word2vec-представлений точно так же оставит нас в окрестности того же самого слова. Как распространить состязательные порождающие сети на дискретные объекты? Это пока остается интересной открытой проблемой.

Глава 9
Глубокое обучение с подкреплением,

или Удивительная история о том, как корейский чемпион сумел победить глубокую сеть в одной партии из пяти

TL;DR

В этой главе мы поговорим о задаче обучения, которое, наверное, больше всего похоже на человеческое, — обучении с подкреплением. Мы:

- познакомимся с мотивацией и общей целью обучения с подкреплением;
- рассмотрим марковские процессы принятия решений и TD-обучение;
- поймем, где здесь нейронные сети, и разберем архитектуру DQN;
- подробно поговорим об одной из самых ярких недавних демонстраций искусственного интеллекта — победе AlphaGo над Ли Седолем;
- изучим методы градиентного спуска по стратегиям и их приложения в робототике и других областях.

9.1. Обучение с подкреплением

> Винни-Пух был всегда не прочь немного подкрепиться, в особенности часов в одиннадцать утра, потому что в это время завтрак уже давно окончился, а обед еще и не думал начинаться.
>
> *А. А. Милн. Винни-Пух и все-все-все*
> *(перевод Б. Заходера)*

Задачи машинного обучения, о которых мы говорили до сих пор, можно было разделить на две основные категории:

1) в задачах *обучения с учителем (supervised learning)* есть набор «правильных ответов», и нужно его продолжить на все пространство или, точнее говоря, на те примеры, которые могут встретиться в жизни, но которые еще не попадались в тренировочной выборке; к этому классу относятся задачи регрессии и классификации, и этим мы занимаемся в этой книге все время, будь то классификация рукописных цифр по набору данных MNIST или машинный перевод на основе параллельных текстов на разных языках;
2) в задачах *обучения без учителя (unsupervised learning)* есть просто набор примеров без дополнительной информации, и задача состоит в том, чтобы понять его структуру, выделить признаки, хорошо ее описывающие; мы подробно рассматривали некоторые задачи обучения без учителя в части III, когда говорили об автокодировщиках и предобучении нейронных сетей, да и вообще, как мы уже много раз видели, одно из главных достоинств глубокого обучения — это именно способность очень умело выделять сложные признаки, раскрывать непростую внутреннюю структуру окружающих нас объектов.

Это весьма классическое разделение, вы встретите его в любом учебнике по машинному обучению. Но разве так учимся мы с вами? Разве так работает обучение в реальной жизни?

Представьте, к примеру, как маленький ребенок учится ходить. Как могло бы это выглядеть в парадигмах обучения с учителем и без него:

1) *обучение с учителем* — родители, которые уже понимают, как ходят люди, выдают ребенку набор данных о том, какие мышцы и в какой последовательности нужно напрячь для того, чтобы сделать шаг; набор должен быть достаточно полным и покрывать все или почти все разные ситуации, в которых ребенок может захотеть сделать шаг, а также, разумеется, должен содержать и отрицательные примеры: как именно он упадет, если будет напрягать мышцы неправильно; ну или, чтобы не доводить до такого уж абсурда, родители просто показывают ребенку несколько десятков тысяч фотографий, на которых изображены разные фазы разных шагов, а дальше его волшебная глубокая сеть выделяет из них признаки и передает ребенку нужную информацию;

2) *обучение без учителя* — ребенок просто наблюдает, как ходят его папа и мама; и потом, постепенно, из сотен тысяч записанных в мозг картинок складываются нужные признаки: если вот так слегка напряглась левая икроножная мышца, а вот этак — правая, в том же кластере будут видео папы и мамы, которые идут вперед ровно и уверенно; ну а если напрячь иначе, то вот, давно уже наготове тысячи картинок того, как папа и мама неуклюже падают, не сумев сделать шаг.

Правда же, совсем как в жизни? Как сейчас помню, мама приходила к моей кроватке и начинала показывать, как надо ходить и как не надо...

На самом деле мы далеко не всегда знаем набор правильных ответов. Часто мы просто делаем то или иное действие и получаем результат. Когда ребенок учится ходить, он сначала делает что-то отдаленно похожее на правду, что подсказывает ему инстинкт[1], и получает результат, поначалу скорее всего отрицательный. Тогда он немного меняет свое поведение, возможно, довольно случайным образом, и смотрит, не увеличилась ли, так сказать, целевая функция; а когда что-то начинает получаться, ребенок запоминает, как это произошло, и затем повторяет то же самое, пытаясь развить успех дальше.

Отсюда и основная идея *обучения с подкреплением (reinforcement learning)*. В этой постановке задачи агент взаимодействует с окружающей средой, предпринимая действия; окружающая среда его поощряет за эти действия, а агент продолжает их предпринимать, пытаясь максимизировать свою «награду» за это; его награда тоже приходит из окружающей среды.

Историю обучения с подкреплением[2] можно смело начинать от работ Павлова[3] об условных и безусловных рефлексах, хотя психологические подходы были известны и ранее, например в работах Александра Бэна середины XIX века [24]. Его основная идея состояла в том, что мы обучаемся методом проб и ошибок: когда какое-нибудь спонтанное (то есть по сути случайное) движение совпадает с состоянием удовольствия, «удерживающая сила духа» устанавливает между ними ассоциацию. Ту же линию продолжили и теории Ллойда Моргана в психологии и Эдварда Ли Торндайка в его трудах об интеллекте животных [530]: полезное действие, вызывающее удовольствие, закрепляется и усиливает связь между ситуацией и реакцией, а вредное, вызывающее неудовольствие, ослабляет связь и исчезает.

---

[1] Впрочем, соотношение между инстинктами и собственно прижизненным обучением — это очень тонкий вопрос; мы не будем на нем подробно останавливаться, но последнего слова здесь наука еще не сказала.

[2] Эту историю мы излагаем по эссе Валентина Малых [593].

[3] *Иван Петрович Павлов* (1849–1936) — русский ученый, первый русский нобелевский лауреат, фактический создатель науки о высшей нервной деятельности. Работа Павлова чем-то похожа на то, как в обучении глубоких сетей сходятся теоретические и практические новшества: его знаменитые опыты с желудочным соком, имеющие огромное теоретическое значение и ставшие основой всей современной физиологии, были бы невозможны без виртуозной техники вивисекции, как это тогда называлось. А из других трудов Павлова возьмем на себя смелость порекомендовать прочесть две лекции с заманчивым названием «Об уме вообще, о русском уме в частности»: сказано в 1918 году, но до сих пор чрезвычайно актуально.

А после того как в психологию пришли бихевиористы, и особенно Б. Ф. Скиннер[1], эта идея стала абсолютным и не вызывающим сомнения мейнстримом.

В целом это и есть идея обучения с подкреплением, и в машинном обучении она тоже появилась очень давно. Еще Тьюринг в своих эссе об искусственном интеллекте описывал архитектуру системы «боли и наслаждения», которая должна управлять тем, что именно сохраняется в памяти искусственного интеллекта [542]. Ранние работы информатиков на эту тему похожи на работы Павлова и Скиннера: например, в 1952 году Клод Шеннон продемонстрировал лабиринт, по которому бегал мышонок по имени Тесей, исследуя его тем самым методом проб и ошибок и запоминая свой путь [490].

В своей диссертации Марвин Минский [362] представил вычислительные модели обучения с подкреплением, а также описал аналоговую вычислительную машину, построенную на элементах, которые он назвал SNARC — стохастический вычислитель, обучаемый через подкрепление и по сути аналогичный нейрону. Эти элементы должны были соответствовать изменяемым семантическим связям в человеческом мозге; обратите внимание, что это совсем не то же самое, что перцептрон Розенблатта, который мы обсуждали в разделе 3.2 и который явным образом обучается с учителем.

Если говорить чуть более формально, на каждом шаге агент может находиться в некотором состоянии $s \in S$, где S — множество всех состояний, и выбирает некоторое действие $a \in A$ из имеющегося набора действий A.

После этого окружающая среда сообщает агенту, какую награду r (от слова *reward*) он за это получил и в каком состоянии s' оказался в результате своих действий. Задача агента — заработать как можно большую награду:

- либо за отведенное время h (от слова horizon, «горизонт»); такая постановка задачи называется *моделью с конечным горизонтом*, и целевую функцию (доход, revenue) можно представить так:

$$R = \mathbb{E}\left[r_0 + r_1 + r_2 + \ldots + r_h\right] = \mathbb{E}\left[\sum_{t=0}^{h} r_t\right],$$

  где r_t — награда агента на шаге t;

[1] *Беррес Фредерик Скиннер* (Burrhus Frederic Skinner, 1904–1990) — американский психолог, бихевиорист и социальный философ. Скиннер довел идеи бихевиоризма до их наивысшего развития; он считал, что все многообразие поведения людей и животных так или иначе сводится к принципам позитивного и негативного подкрепления. Для развития своих идей он изобрел пресловутый Skinner box, в котором мышка может нажать на специальный рычажок и получить за это действие ту или иную награду. Во время Второй мировой войны Скиннер предложил конструкцию ракеты, управляемой специально натренированным голубем (проект был многообещающий, но, к сожалению, так и не реализовался до конца), а затем обнаружил у голубей даже суеверия: если давать голубю еду в случайные моменты времени, голубь очень постарается заметить случайные совпадения того, что он делал в эти моменты, с поступлением еды, и будет пытаться повторять эти случайные действия.

- либо за все бесконечное предстоящее время; в таком случае, конечно, просто суммировать не получится, но легко понять, что в таких задачах почти всегда получить награду раньше выгоднее, чем позже[1], поэтому обычно в целевую функцию модели с *бесконечным горизонтом* вводят некоторую константу (discount factor), на которую вознаграждение уменьшается с каждым последующим шагом:

$$R = \mathbb{E}\left[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots + \gamma^k r_k + \ldots\right] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right].$$

Есть и третий вариант, *модель среднего вознаграждения (average-reward model)*, когда оптимизируется $\lim_{h\to\infty} \mathbb{E}\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right]$, но мы ее использовать практически не будем. А конечный горизонт легко свести к бесконечному: достаточно поставить $\gamma = 1$ и считать, что за горизонтом, после времени h, все награды r_t, $t > h$, равны нулю независимо от действий агента.

Самая простая постановка задачи обучения с подкреплением — это так называемая задача о *многоруких бандитах (multiarmed bandits)*. Формально здесь все точно так же, но $|S| = 1$, то есть состояние агента не меняется. У него просто есть некий фиксированный набор действий A и возможность выбирать из этого набора действий. Такая модель называется задачей о многоруких бандитах, потому что ее легко представить себе так: агент находится в комнате с несколькими игровыми автоматами, у каждого автомата свое ожидание выигрыша, а агенту нужно выиграть как можно больше денег, бросая монетки то в один автомат, то в другой.

Получается, что агент сам платит за свое обучение, и ему нужно суметь вовремя понять, что обучение (исследование, exploration) можно заканчивать и переходить к использованию полученных знаний (exploitation). *Exploration vs. exploitation* — это главная проблема, основная дилемма задачи о многоруких бандитах.

Мы не будем подробно рассказывать о многоруких бандитах, потому что глубокое обучение с подкреплением обычно начинается все-таки там, где состояний несколько, но пару важных замечаний сделать нужно. Начнем с простейшего *жадного алгоритма*, который всегда выбирает стратегию, максимизирующую прибыль. Прибыль можно оценить как среднее вознаграждение, полученное от того или иного действия:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a}.$$

Здесь и далее мы будем предполагать, что мы сначала дергаем по разу за каждую ручку, чтобы нигде не делить на ноль, а потом уже начинаем запускать алгоритмы.

---

[1] Представьте себе, например, награду в виде денег: сто рублей сегодня точно лучше, чем сто рублей через год, хотя бы потому, что их можно положить в банк и через год получить больше ста рублей, не говоря уж о более выгодных вложениях.

Жадный алгоритм кажется достаточно простым и логичным. Что же с ним не так? Дело в том, что оптимум легко проглядеть, если на начальной выборке нам немножко не повезет, что более чем возможно. Представьте себе, например, *бинарных* бандитов, у которых выплаты всегда равны нулю или единице. Тогда жадный алгоритм буквально *никогда* не вернется к тем ручкам, которые на первом обходе выдали ноль, ведь их среднее будет нулевым, а у тех ручек, которые хоть раз выдали единицу, нуля уже никогда не получится.

Поэтому полезная эвристика в задаче о многоруких бандитах, да и во всем обучении с подкреплением — *оптимизм при неопределенности*. Это значит, что выбирать нужно жадно, но при этом прибыль оценивать оптимистично, и всегда требовать серьезные свидетельства, чтобы перестать проверять ту или иную стратегию.

Одним из ключевых теоретических инструментов в обучении с подкреплением является *ϵ-жадная стратегия* (ϵ-greedy): выбрать действие с наилучшей ожидаемой прибылью с вероятностью $1 - \epsilon$, а с вероятностью ϵ выбрать случайное действие. Такая стратегия приводит к тому, что алгоритм не отличает хорошую альтернативу от бесполезной, выделяя только лучшую, но все равно процесс исследования получается разумным, и про него обычно можно доказать разные полезные теоретические свойства, ведь ϵ-жадная стратегия означает, что мы всегда можем дернуть за каждую ручку с положительной константной вероятностью. На практике обычно начинают с больших ϵ, а затем уменьшают; выбор стратегии этого уменьшения — важный параметр алгоритма.

Другой естественный способ применить оптимистично–жадный метод — это известные из статистики *доверительные интервалы*. Давайте для каждого действия хранить статистику числа таких действий n и числа успешных действий w (или среднего вознаграждения), а потом для выбора ручки, за которую хочется дернуть, использовать *верхнюю границу* доверительного интервала вероятности успеха (или ожидания вознаграждения). Тем самым мы достигнем как раз нужного эффекта: сначала все доверительные интервалы будут очень широкими, и их верхние границы будут очень высоко, а потом, по мере накопления опыта, интервалы начнут сужаться, причем менее исследованные альтернативы будут получать преимущество в выборе. Такая стратегия сойдется к оптимальной ручке, когда доверительные интервалы всех остальных ручек будут полностью лежать ниже ее среднего. Например, для бинарных бандитов, то есть фактически для подбрасывания монетки, с вероятностью 0,95 среднее лежит в интервале $\left(\bar{x} - 1{,}96\frac{s}{\sqrt{n}}, \bar{x} + 1{,}96\frac{s}{\sqrt{n}}\right)$, где 1,96 берется из распределения Стьюдента, n — число испытаний, $s = \sqrt{\frac{\sum(x-\bar{x})^2}{n-1}}$. Брать верхнюю границу доверительного интервала — отличный метод, если вероятностные предположения соответствуют действительности, что не всегда очевидно.

Другой, более простой вариант оптимизма при неопределенности — начать с оптимистичных значений средних: давайте выставим $Q_0(a)$ такими большими, что любое реальное вознаграждение будет «разочаровывать» нас, но не слишком

большими – нам нужно, чтобы достаточно быстро Q_0 усреднилось с реальными оценками средних. Тогда можно использовать тривиальную жадную стратегию, и она даст тот же эффект: сначала средние у всех ручек будут заведомо слишком высокими, а потом по мере накопления опыта уменьшатся и будут постепенно сходиться к истинным средним, причем чем меньше было экспериментов, тем большее влияние оказывает Q_0 на среднее. Сами значения Q_0 можно менять и тем самым управлять балансом между exploration и exploitation.

Современные алгоритмы действуют примерно по той же общей схеме: они присваивают приоритет каждой ручке i и доказывают оценки непосредственно на цену обучения (regret). Так, стратегия UCB1 [13] учитывает неопределенность, «оставшуюся» в той или иной ручке, и старается ограничить цену обучения так: если из n экспериментов n_i раз дернули за i-ю ручку и получили среднюю награду $\hat{\mu}_i$, алгоритм UCB1 присваивает ей приоритет

$$\text{Priority}_i = \hat{\mu}_i + \sqrt{\frac{2\log n}{n_i}}.$$

Дергать дальше надо за ручку с наивысшим приоритетом. В [13] доказано, что при таком подходе субоптимальные ручки будут дергать $O(\log n)$ раз, и цена обучения будет составлять $O(\log n)$. А меньше $O(\log n)$ и не получится — есть соответствующая нижня оценка; впрочем, константы тут тоже важны, так что на UCB1 человеческая мысль не остановилась, но мы в это уже вдаваться не будем.

И последнее общее замечание, которое нам пригодится ниже. Давайте посмотрим на то, как пересчитывать оценки среднего $Q_t(a) = \frac{r_1+\ldots+r_{k_a}}{k_a}$ при поступлении новой информации. В этом, конечно, ничего сложного нет — будем хранить число попыток k и пересчитывать как

$$Q_{k+1} = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i = \frac{1}{k+1}\left[r_{k+1} + \sum_{i=1}^{k} r_i\right] =$$
$$= \frac{1}{k+1}\left(r_{k+1} + kQ_k\right) = Q_k + \frac{1}{k+1}\left(r_{k+1} - Q_k\right).$$

У нас получилась очень важная формула, частный случай общего правила — сдвигаем оценку так, чтобы уменьшалась ошибка:

$$\text{НоваяОценка} := \text{СтараяОценка} + \text{Шаг}\left[\text{Цель} - \text{СтараяОценка}\right].$$

Именно так выглядит, например, общее правило градиентного спуска: производная указывает направление на цель в текущей точке, а шаг — это скорость обучения.

Заметим теперь, что шаг у среднего не постоянный, а уменьшающийся со временем: из формулы $Q_{k+1} = Q_k + \frac{1}{k+1}\left(r_{k+1} - Q_k\right)$ получается, что шаг ручки a

в момент времени k (после k экспериментов) $\alpha_k(a)$ равен $\frac{1}{k_a+1}$. Изменяя последовательность шагов, можно добиться других эффектов. Например, часто бывает, что выплаты от разных ручек на самом деле нестационарны, то есть меняются со временем. В такой ситуации имеет смысл давать большие веса свежей информации, а далекому прошлому — маленькие. Как это сделать? Можно просто поставить вместо затухающих весов постоянные: у правила обновления $Q_{k+1} = Q_k + \alpha\,[r_{k+1} - Q_k]$ с постоянным $\alpha_k(a) = \alpha$ веса фактически затухают экспоненциально:

$$Q_k = Q_{k-1} + \alpha\,[r_k - Q_{k-1}] = \alpha r_k + (1-\alpha)Q_{k-1} =$$
$$= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2 Q_{k-2} = (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i.$$

Такое правило обновления не сходится (разности между соседними Q_k и Q_{k-1} не обязательно стремятся к нулю с ростом k), но это и хорошо – мы хотим следовать за целью. Есть и общий результат: правило обновления сходится, если последовательность весов не сходится сама, $\sum_{k=1}^{\infty} \alpha_k(a) = \infty$, но сходится в квадрате, $\sum_{k=1}^{\infty} \alpha_k^2(a) < \infty$. И в целом, такой взгляд на вещи сразу объединяет и упорядочивает массу разных методов.

Сами по себе многорукие бандиты используются в задачах, где нужно сравнить несколько альтернатив между собой при помощи экспериментов. Например, несколько лет назад в IT-сообществе были очень популярны статьи, пропагандирующие использование многоруких бандитов для A/B тестирования: каждая альтернатива становится ручкой, и выходит, что размер выборки подбирается автоматически. Можно аналогично использовать бандитов и для оптимизации гиперпараметров, например в тех же глубоких нейронных сетях (или где угодно). Но для нас сейчас это скорее просто первый шаг, упрощенная постановка общей задачи обучения с подкреплением... так давайте же скорее избавимся от упрощений.

9.2. Марковские процессы принятия решений

> Но как я могу принять решение, – спрашивал себя этот рассудительный государь, – если мне совершенно неизвестны те доводы, которые могут привести обе стороны?
>
> *Стендаль. Чрезмерная благосклонность губительна*

Теперь, когда мы поняли основную суть обучения с подкреплением и разобрались с самой простой ситуацией, многорукими бандитами, пора обобщать дальше. В реальности, конечно, часто бывает, что агент не возвращается в точно такое же состояние для нового «хода»: например, играющая в го или шахматы программа должна,

пожалуй, учитывать, что позиция на доске после ее хода и ответа противника слегка изменится. Поэтому теперь нам нужно определить некий процесс, в котором агент последовательно переходит из состояния в состояние в зависимости от своих действий, иногда, в некоторых состояниях, получая награды; все зависимости здесь, конечно, будут не прямыми, а стохастическими, вероятностными.

Здесь возникает вторая центральная дилемма обучения с подкреплением, задача *распределения вознаграждения* (credit assignment): пусть даже мы знаем, выиграли мы партию или проиграли, но какой именно ход привел к победе или поражению? Эта проблема тоже была очевидна с первых шагов теории обучения с подкреплением, ее рассматривал еще Марвин Минский в начале 1960-х годов [363], но успешно решить ее бывает сложно до сих пор.

Итак, пора ввести основное определение данной главы. *Марковский процесс принятия решений* (Markov decision process) [233, 519] состоит из:

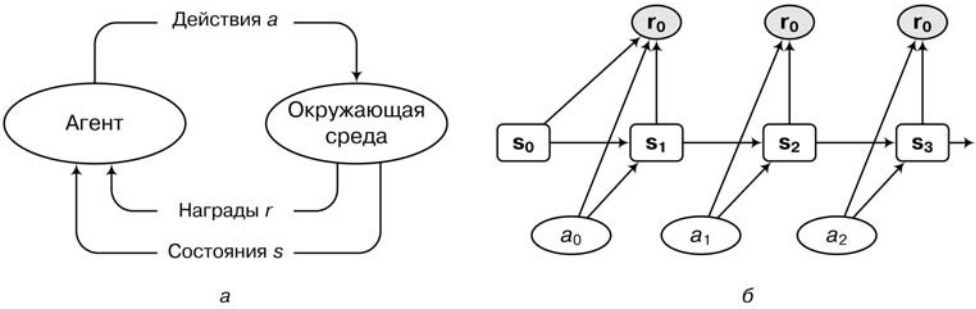
- множества состояний S;
- множества действий A;
- функции вознаграждения $R : S \times A \to \mathbb{R}$; это значит, что ожидаемое вознаграждение при переходе из s в s' после действия a составляет $R^a_{ss'}$;
- функции перехода между состояниями $p^a_{ss'} : S \times A \to \Pi(S)$, где $\Pi(S)$ — множество распределений вероятностей над S; это значит, что вероятность попасть из состояния s в состояние s' после действия a равна $P^a_{ss'}$.

Мы проиллюстрировали марковский процесс принятия решений на рис. 9.1: слева, на рис. 9.1, *а*, вы видите самую общую, классическую схему обучения с подкреплением. А на рис. 9.1, *б* показана более подробная схема марковского процесса принятия решений, где видно, какие его части от каких зависят.

Процесс называется *марковским*, потому что вероятности переходов между состояниями не зависят от истории предыдущих переходов; вообще, слово «марковский» в математике и информатике всегда обозначает именно это: отсутствие памяти, независимость от того, что было в прошлом. Это кажется очень сильным предположением, но на самом деле оно довольно часто выполняется. Например, в го или шахматах неважно, какими ходами мы пришли к текущей позиции, важна лишь позиция сама по себе.

А если марковское свойство существенно нарушено, то часто можно просто записать то, что нужно «помнить» из прошлого, в качестве части определения состояния, и тогда марковское свойство будет восстановлено. Например, состояние при игре в покер должно включать в себя не только текущий размер ставок, но и историю ставок в текущей раздаче, а возможно, и более долгую историю взаимодействия между игроками[1].

[1] Кстати, покер — это отдельная и очень сложная история, во многие его разновидности программы как раз пока что играют довольно плохо, и важная часть проблемы именно в том, чтобы правильно помнить и понимать историю взаимодействий. Однако стоит отметить модель DeepStack [113], которая научилась хорошо играть в heads-up no-limit Texas hold'em, то есть в безлимитный покер против одного оппонента.



а *б*

Рис. 9.1. Марковский процесс принятия решений: *а* — общая схема обучения с подкреплением, *б* — подробная схема марковского процесса принятия решений

Здесь стоит сделать небольшое отступление. Общая постановка задачи обучения с подкреплением похожа на постановку задачи *теории оптимального управления*, в котором известна некоторая модель объекта управления, на объект есть некоторые воздействия, и задача состоит в том, чтобы найти оптимальные воздействия, которые максимизируют нужную целевую функцию. Теория оптимального управления — область старая и заслуженная, в ней появились и уравнения Беллмана, о которых речь пойдет ниже, и принцип максимума Понтрягина[1]. И объекты в ней рассматриваются куда сложнее, чем обычно в обучении с подкреплением. Действительно, многое из того, о чем мы будем говорить в этой главе, можно рассматривать как часть теории управления. Однако акценты расставлены немного иначе: в обучении с подкреплением основная сложность не в том, чтобы найти оптимальное управление, а в том, чтобы изучить окружающую среду — если мы среду и объект управления уже знаем (в теории оптимального управления это называется *идентификацией системы*), обычно найти оптимальное действие не так уж сложно. Но области действительно взаимопроникающие, и изучить методы теории управления будет очень полезно, если вы всерьез решите заняться обучением с подкреплением.

[1] *Лев Семенович Понтрягин* (1908–1988) — советский математик, один из лучших математиков XX века. Когда Льву было 14 лет, возле него взорвался примус, и после неудачной операции Понтрягин полностью потерял зрение. Это предопределило судьбу семьи Понтрягиных: отец в результате потерял трудоспособность и вскоре умер от инсульта, а мать посвятила себя сыну и его математическому образованию. Она даже выучила немецкий язык, чтобы читать Льву статьи на языке тогдашней науки. Однако слепота не помешала ни активной жизни, ни тем более научной карьере Понтрягина. Он начал с топологии, получил ряд блестящих результатов, но позже вспоминал: «Я не мог ответить на вопрос, для чего нужно все это, все то, что я делаю? Самая пылкая фантазия не могла привести меня к мысли, что гомологическая теория размерности может понадобиться для каких-нибудь практических целей». Поэтому Понтрягин занялся прикладной математикой: его четырехстраничная статья 1937 года «Грубые системы» (с А. А. Андроновым) заложила основы теории динамических систем, позже развил теорию дифференциальных игр, а принцип максимума Понтрягина остается основополагающим принципом теории оптимального управления.

Подробный пример марковского процесса принятия решений, к которому мы будем постоянно возвращаться в этом разделе, приведен на рис. 9.2. На рисунке прямоугольники соответствуют состояниям, белые овалы — возможным действиям, а овалы со штриховкой — вознаграждениям, получаемым на этом переходе между состояниями. Сам процесс соответствует простой игре «чет-нечет», проходящей в два круга. Правила игры таковы:

- в каждом круге и мы, и противник выбираем число; если оба числа четные или оба нечетные, мы выигрываем; если четность разная (одно число четное, другое нечетное), мы проигрываем;
- в первом круге выигрыш и проигрыш равен $+1$ и -1 соответственно, а во втором круге ставки повышаются, и выигрыш и проигрыш начинают стоить $+5$ и -5 соответственно;
- после второго круга игра заканчивается (поэтому мы не стали выделять много состояний после второго круга, ограничившись двумя для удобства);
- поскольку игра конечная, процесс получается эпизодический, и $\gamma = 1$.

Но это еще не все; чтобы полностью задать марковский процесс принятия решений, нужно еще определить вероятности переходов между состояниями; они показаны на стрелках, соответствующих переходам. Здесь будет небольшое усложнение, без которого игра была бы совсем скучной. противник будет не просто подбрасывать монетку, а действовать таким образом:

- в первом круге противнику больше нравятся нечетные числа: вероятность четного числа от него равна $\frac{1}{3}$, а нечетного — $\frac{2}{3}$;
- а во втором круге он смотрит на то, к чему привела его игра в первый раз: если он в первом круге выиграл, то он считает, что сыграл правильно, и повторяет свой выбор, а если проиграл, возвращается к «стратегии по умолчанию» и во втором круге просто подбрасывает монетку.

Не поленитесь и проследите все стрелочки на рис. 9.2: пример еще пригодится.

Теперь, когда у нас есть состояния и переходы между ними, придется научиться различать *функцию вознаграждения* (reward function, непосредственное подкрепление, то, что мы обозначили за R) и то, что мы назовем *функцией значения состояния* (value function, $V(s)$); это будет общее ожидаемое подкрепление, которое можно получить, если начать с этого состояния. Суть многих методов обучения с подкреплением – в том, чтобы оценивать и оптимизировать функцию значений; по сути задача наша сводится к тому, чтобы выбирать ходы, которые приводят к состоянию с максимальным значением $V(s)$. Для марковских процессов можно формально определить:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right],$$

где π — это стратегия, которой следует агент.

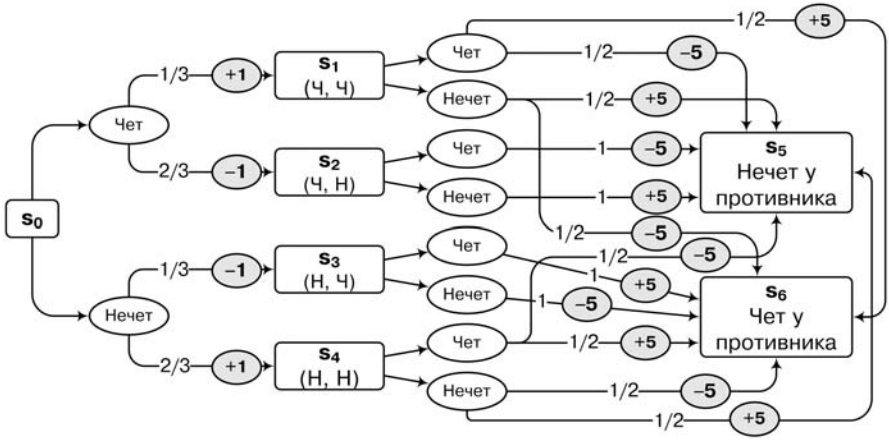


Рис. 9.2. Пример марковского процесса принятия решений: игра «чет-нечет»

Обратите внимание, что слово *стратегия* (policy) здесь понимается в достаточно строгом смысле. Поскольку, вообще говоря, агент может подбрасывать монетки, чтобы выбирать очередные действия (и многие стратегии из раздела 9.1 так и делали, например часто встречающаяся ϵ-жадная стратегия), стратегия π — это функция, которая для данного состояния s выдает распределение вероятностей на множестве действий A. Мы также будем обозначать через $\pi(a,s)$ вероятность выбрать действие $a \in A$ в состоянии s, а если захотим подчеркнуть, что стратегия задана параметрически с вектором параметров $\boldsymbol{\theta}$, будем писать $\pi(a,s;\boldsymbol{\theta})$ (это нам особенно пригодится в разделе 9.5). Если же стратегия детерминированная, это просто значит, что для каждого s все вероятности $\pi(a,s)$ равны нулю, кроме одной, которая равна единице.

Впрочем, функция значений состояния все еще удалена от непосредственных решений агента, ведь он не может просто взять и выбрать следующее состояние. Игрок в го не может сам определить позицию перед своим следующим ходом, она будет зависеть и от хода противника. Поэтому ожидаемое в будущем подкрепление часто рассматривают более детально: функция Q выражает общее подкрепление, ожидаемое, если агент начнет в состоянии s и сделает там действие a:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t \mid s_t = s, a_t = a\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right].$$

Функции V и Q — это как раз то, что нам нужно оценить; если бы мы их знали, можно было бы просто выбирать то действие a, которое максимизирует $Q(s,a)$.

Давайте попробуем подсчитать функцию значений состояния $V^\pi(s)$, последовательно разворачивая ее определение. В цепочке равенств ниже мы сначала выписываем определение ожидаемого суммарного вознаграждения с бесконечным горизонтом, затем отделяем от него первый шаг и замечаем, что после него остается точно такое же выражение, просто умноженное на γ:

$$
\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi\left[R_t \mid s_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+1} \mid s_t = s\right] = \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \mid s_t = s\right] = \\
&= \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}\left(R^a_{ss'} + \gamma\mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k+2} \mid s_{t+1} = s'\right]\right) = \\
&= \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}\left(R^a_{ss'} + \gamma V^\pi(s')\right).
\end{aligned}
$$

У нас получилось, что для известной стратегии π значения V^π удовлетворяют так называемым *уравнениям Беллмана*[1].

Уравнения Беллмана — это по сути математическое выражение принципа динамического программирования. Такие уравнения и их аналоги появляются во множестве разных приложений. А в нашем случае получается, что, теоретически говоря, для того чтобы найти значения $V^\pi(s)$, можно просто взять и решить систему линейных уравнений, неизвестными в которых являются $V^\pi(s)$ для разных состояний $s \in S$.

Правда, для этого нужно знать все параметры марковского процесса, то есть функции R и P, а с этим в реальных ситуациях плохо. Все, что у нас обычно есть на входе, — это окружающая среда, выдающая награды как черный ящик.

Так что в реальных задачах функции R и P тоже приходится обучать по ходу дела. На самом деле методы обучения с подкреплением делятся на те, которые обучают функции R и P в явном виде, и те, которые обходятся без этого и сразу обучают V и/или Q.

---

[1] *Ричард Беллман* (Richard Ernest Bellman, 1920–1984) — американский математик и инженер. Его жизненный путь был очень интересным, но достаточно обычным для ученого того времени: родился в Бруклине (кстати, у Беллмана российские и польские корни), во время Второй мировой войны работал над Манхэттенским проектом как теоретический физик, а с 1949 года поступил на работу в RAND Corporation, знаменитый аналитический центр (think tank), изначально организованный Douglas Aircraft Company для нужд армии, но быстро превратившийся в некоммерческую научную организацию, занимающуюся самыми разными научными и инженерными вопросами. Именно там Беллман разработал основы теории марковских процессов принятия решений и динамического программирования в целом — кстати, он сам придумал это название, мотивировав его тем, что «прилагательное dynamic было невозможно использовать в негативном смысле... это было такое название, что даже конгрессмен не возразил бы».

Но для простых примеров воспользоваться уравнениями Беллмана вполне возможно. Давайте попробуем подсчитать функцию значений состояния для какой-нибудь стратегии в примере на рис. 9.2. Например, пусть стратегия π — это просто равновероятный выбор на каждом шаге, подбрасывание честной монетки. Тогда:

$$V^\pi(s_0) = \frac{1}{2}\left(\frac{1}{3}\left(1 + V^\pi(s_1)\right) + \frac{2}{3}\left(-1 + V^\pi(s_2)\right)\right) + \\ + \frac{1}{2}\left(\frac{1}{3}\left(-1 + V^\pi(s_3)\right) + \frac{2}{3}\left(1 + V^\pi(s_4)\right)\right).$$

Считаем дальше. Теперь рассмотрим, к примеру, состояние s_1:

$$V^\pi(s_1) = \frac{1}{2}\left(\frac{1}{2}\left(-5 + V^\pi(s_5)\right) + \frac{1}{2}\left(5 + V^\pi(s_6)\right)\right) + \\ + \frac{1}{2}\left(\frac{1}{2}\left(5 + V^\pi(s_5)\right) + \frac{1}{2}\left(-5 + V^\pi(s_6)\right)\right).$$

Поскольку после второго круга игра заканчивается, $V^\pi(s_5) = V^\pi(s_6) = 0$, а значит, $V^\pi(s_1) = 0$ тоже. Легко видеть, что для остальных состояний это тоже верно: $V^\pi(s_2) = V^\pi(s_3) = V^\pi(s_4) = 0$, а значит, и $V^\pi(s_0) = 0$.

Итак, подбрасывая монетку на каждом круге, мы ничего не выиграем и ничего не проиграем. Оптимальная ли это стратегия или можно все-таки что-то выиграть? Давайте попробуем ответить на этот вопрос.

Но давайте сначала для простоты предположим, что мы уже точно знаем нашу модель. Задача — найти оптимальную стратегию поведения для агента в этой модели. В такой постановке мы уже умеем искать $V^\pi(s)$, решая уравнения Беллмана, но разных стратегий еще больше, чем состояний, и перебрать их мы обычно, простите за каламбур, не в состоянии.

К счастью, это и не обязательно: нам нужно только научиться подсчитывать *оптимальное значение состояния*, то есть искать ожидаемую суммарную прибыль, которую получит агент, если начнет с этого состояния и будет следовать оптимальной стратегии:

$$V^*(s) = \max_\pi E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right].$$

Эту функцию можно по тем же причинам определить как решение уравнений:

$$V^*(s) = \max_a \sum_{s' \in S} P^a_{ss'}\left(R^a_{ss'} + \gamma V^*(s')\right)$$

(они тоже называются уравнениями Беллмана), а затем выбрать оптимальную стратегию исходя из подсчитанных значений V^*:

$$\pi^*(s) = \arg\max_a \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^*(s') \right).$$

Как решать уравнения? Они уже не линейные, и решить их точно и эффективно не получится, но наше дело все равно отнюдь не безнадежно. Как известно, если сложное уравнение представлено в виде $x = f(x)$, то его можно решать итеративно методом Ньютона: начать с какого-нибудь x_0 и последовательно пересчитывать $x_{k+1} = f(x_k)$, пока процесс не сойдется, то есть пока изменения $|x_{k+1} - x_k|$ не станут совсем маленькими. Здесь эта идея великолепно работает, ведь, как легко заметить, уравнения уже представлены в нужном виде!

Это можно делать и для исходных линейных уравнений, получится быстрее, чем решать систему «по-честному». Естественно, в результате получается приближенный, численный метод решения уравнений Беллмана, но в машинном обучении нам ничего другого и не требуется.

Так что, чтобы подсчитать функции значений состояний для данной стратегии π, можно просто итеративно пересчитывать их по уравнениям Беллмана:

$$V^\pi(s) := \sum_a \pi(s,a) \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^\pi(s') \right),$$

пока процесс не сойдется.

А для оптимальных значений мы будем пересчитывать уравнения с максимумами вместо математических ожиданий:

$$V^*(s) := \max_a \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma V^*(s') \right).$$

Ровно то же самое можно сделать и для функции $Q(s,a)$, последовательно повторяя вычисления по следующей формуле:

$$Q(s,a) := \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma \sum_{a'} \pi(s,a') Q(s,a') \right),$$

до сходимости. И с оптимальным $Q^*(s,a)$ все обстоит точно так же: повторяем

$$Q^*(s,a) := \sum_{s' \in S} P^a_{ss'} \left(R^a_{ss'} + \gamma \max_{a'} Q^*(s,a') \right),$$

пока не сойдется; кстати, потом можно вычислить оптимальную функцию значений как $V^*(s) := \max_a Q^*(s,a)$.

Давайте подсчитаем оптимальные значения $V^*(s)$ и $Q^*(s,a)$ в примере, изображенном на рис. 9.2. Снова начинаем сначала:

$$V^\pi(s_0) = \max\left\{\frac{1}{3}\left(1 + V^\pi(s_1)\right) + \frac{2}{3}\left(-1 + V^\pi(s_2)\right),\right.$$
$$\left.\frac{1}{3}\left(-1 + V^\pi(s_3)\right) + \frac{2}{3}\left(1 + V^\pi(s_4)\right)\right\}.$$

Игра по-прежнему заканчивается в s_5 и s_6, так что $V^*(s_5) = V^*(s_6) = 0$. Воспользуемся этим, чтобы подсчитать V^* от состояний после первого круга игры:

$$V^*(s_1) = \max\left\{\frac{1}{2}(-5) + \frac{1}{2}(5), \frac{1}{2}(5) + \frac{1}{2}(-5)\right\} = 0.$$

Аналогично и $V^*(s_4) = 0$. А вот для s_2 и s_3 теперь ситуация более оптимистическая:

$$V^*(s_2) = \max\{-5, 5\} = 5, \quad V^*(s_3) = \max\{5, -5\} = 5.$$

Подставляя это в выражение для $V^*(s_0)$, получим:

$$V^*(s_0) = \max\left\{\frac{1}{3} + \frac{2}{3}(-1+5), \frac{1}{3}(-1+5) + \frac{2}{3}\right\} = 3.$$

Смотрите-ка — мы можем выиграть, причем с немалым в среднем счетом! Чтобы узнать, как выиграть, нужно подсчитать функцию Q^*; сделаем это для начального состояния s_0, подставляя остальные значения сразу (они считаются очевидным образом):

$$Q^*(s_0,\text{Чет}) = \frac{1}{3}(1 + \max_a Q^*(s_1,a)) + \frac{2}{3}(-1 + \max_a Q^*(s_2,a)) = \frac{1}{3}(1+0) + \frac{2}{3}(-1+5) = 3,$$
$$Q^*(s_0,\text{Нечет}) = \frac{1}{3}(-1 + \max_a Q^*(s_3,a)) + \frac{2}{3}(1 + \max_a Q^*(s_4,a)) = \frac{1}{3}(-1+5) + \frac{2}{3}(1+0) = 2.$$

Это значит, что на первом круге нужно выбирать не имеющее большую вероятность немедленной победы действие Нечет (мы же знаем, что противник любит нечетные числа), а наоборот: нужно сначала поддаться противнику, чтобы успокоить его и заставить повторить свое действие — тогда-то мы его и достанем, а второй круг куда важнее первого. И весь этот хитрый план получился естественным образом из уравнений Беллмана.

Мы проиллюстрировали некоторые результаты наших вычислений на рис. 9.3: серым на нем показаны те состояния и действия, в которые мы никогда не попадем, если будем следовать оптимальной стратегии, задаваемой Q^*, а черным — те, в которые попасть можем (в состоянии s_1 было все равно, какое действие выбирать, так что выбрали Чет).

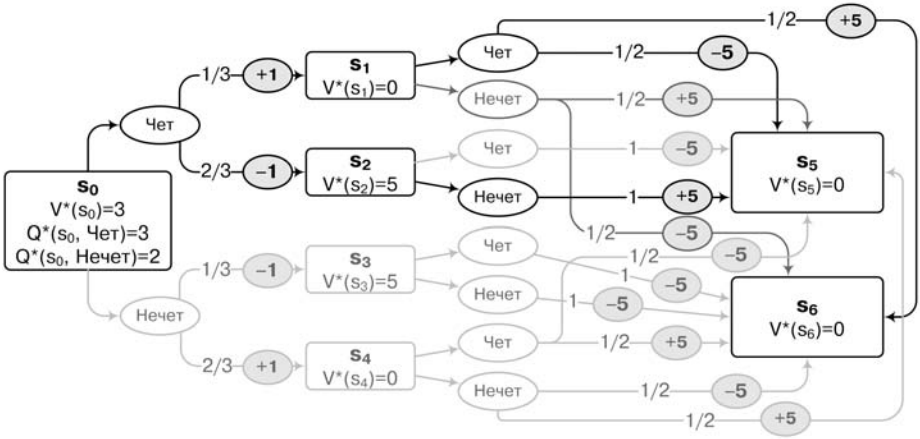


Рис. 9.3. Игра «чет-нечет»: значения функции состояний и оптимальная стратегия

Заметим, что пересчет в нашем алгоритме использует информацию от всех возможных состояний-предшественников; поскольку состояний обычно очень много, все эти формулы пока что фактически неприменимы на практике. Но можно сформулировать аналогичное правило и для одного «тренировочного примера», состоящего из текущего состояния s, произведенного действия a, состояния s', в которое мы после этого перешли, и непосредственной награды r:

$$Q(s,a) := Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right).$$

Теоретические гарантии у этого метода появляются, если каждая пара (s,a) встречается в процессе обучения бесконечное число раз, s' выбирают из распределения $P^a_{ss'}$, а r сэмплируется со средним $R(s,a)$ и ограниченной дисперсией.

Впрочем, на самом деле наша цель — не функции V и Q, а оптимальная стратегия π^*, и ищем мы V^π только затем, чтобы улучшить π. Как это можно сделать?

Итак, мы достаточно долго уже рассуждали о функциях значений состояний и пар состояние–действие, многое поняли, а теперь пора переходить к самому вкусному. Практически все современные подходы к обучению основаны на очень простом, но очень мощном принципе, который называется *TD-обучение* (TD-learning), от слов *temporal difference* (временные разности). Общий принцип TD-обучения таков: давайте обучать состояния на основе уже обученных нами оценок для последующих состояний. Каждый раз, когда мы делаем очередной переход, мы немножко «подтягиваем» функцию V для того состояния (или функцию Q для пары состояние-действие), из которого мы вышли, к значению функции V для того состояния (или функции Q для пары состояние-действие), в которое мы попали.

Простейший алгоритм TD-обучения, так называемое $TD(0)$-обучение, выглядит так. Сначала нужно инициализировать функцию $V(s)$ и стратегию π произвольно (обычно как-то случайно), а затем на каждом эпизоде обучения:

- инициализировать s;
- для каждого шага в эпизоде:
  - выбрать a по стратегии π;
  - сделать a, пронаблюдать результат r и следующее состояние s;
  - обновить функцию V на состоянии s по формуле:

$$V(s) := V(s) + \alpha\left(r + \gamma V(s') - V(s)\right);$$

  - перейти к следующему шагу, присвоив $s := s'$.

Вот и все! На первый взгляд кажется, что здесь происходит какая-то черная магия: мы обучаем $V(s)$ на основе других значений $V(s')$... но их мы тоже инициализировали случайным образом! Однако все работает: смысл в том, чтобы использовать уже обученные закономерности для поиска более глубоких закономерностей. Сначала обучатся значения $V(s)$ на состояниях, которые непосредственно ведут к настоящим наградам r, а потом эти значения будут постепенно передавать накопленные в них «знания» дальше, к предыдущим состояниям. В результате обучение получится целенаправленным, TD-обучение гораздо быстрее и эффективнее, чем другие стратегии.

Впрочем, и его можно улучшить. $TD(0)$ смотрит на один шаг вперед; можно рассмотреть алгоритм, который будет обновлять состояния сразу на много шагов назад. Он называется $TD(\lambda)$, и λ здесь играет ту же роль, что и раньше: мы обновляем каждое состояние u по формуле:

$$V(u) := V(u) + \alpha\left(r + \gamma V(s') - V(s)\right)\epsilon(u)$$

на основе значения $\epsilon(u)$ (от слова *eligibility*), которое показывает, насколько часто это состояние посещалось в прошлом. Значения $\epsilon(u)$ точно так же экспоненциально затухают с показателем λ:

$$\epsilon(s) = \sum_{k=1}^{t} \lambda^{t-k}[s = s_k],$$

где $[s = s_k]$ равно единице, если $s = s_k$, и нулю в противном случае.

Если $\lambda = 0$, $TD(\lambda)$ превращается в уже знакомый нам $TD(0)$. А значения $\epsilon(u)$ можно тоже хранить и пересчитывать в реальном времени после каждого нового перехода:

$$\epsilon(u) := \begin{cases} \lambda\epsilon(u) + 1, & \text{если текущее состояние} - \text{это } u, \\ \lambda\epsilon(u) & \text{в противном случае.} \end{cases}$$

Конечно, на практике обновляют не все состояния, а несколько с самыми большими значениями $\epsilon(u)$, которые в реальной реализации обычно хранятся в приоритетной очереди.

В реальный алгоритм принцип TD-обучения можно превратить разными способами. Если реализовать его для функции Q совсем в лоб, получится алгоритм SARSA, который представляет собой on-policy TD-обучение, после каждого очередного перехода $(s_t,a_t,r_{t+1},s_{t+1},a_{t+1})$[1] делает следующее обновление:

$$Q(s_t,a_t) := Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right).$$

Аналогично, SARSA(λ) обновляет значения на всех парах (s,a) с учетом $\epsilon(s,a)$:

$$\begin{aligned}
Q(s,a) &:= Q(s,a) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right]\epsilon_t(s,a),\\
\epsilon_t(s,a) &:= \gamma\lambda\epsilon_{t-1}(s,a) + [s = s_t, a = a_t].
\end{aligned}$$

При этом, правда, стратегия должна быть мягкой, например ϵ-жадной с уменьшающимся ϵ, чтобы алгоритм мог исследовать новые возможные действия, но со временем она должна как-то плавно сходиться; это вносит дополнительные инженерные сложности в реализацию, потому что от выбора характера затухания ϵ или другого параметра «нежадности» может многое зависеть.

Поэтому более популярно off-policy TD-обучение функции Q, которое обычно так и называется *Q-обучением* [559]. Здесь мы сразу решаем уравнения Беллмана относительно максимумов:

$$Q(s_t,a_t) := Q(s_t,a_t) + \alpha\left(r_{t+1} + \gamma\max_a Q(s_{t+1},a) - Q(s_t,a_t)\right).$$

Теперь Q напрямую аппроксимирует оптимальную функцию Q^*, независимо от стратегии; это значит, что мы можем придерживаться абсолютно любой стратегии, а обучаться все равно будут правильные оптимальные значения Q^*. Аналогично можно определить и Q(λ):

$$\begin{aligned}
Q(s,a) &:= Q(s,a) + \alpha\left(r_{t+1} + \gamma\max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\right)\epsilon_t(s,a),\\
\epsilon_t(s,a) &:= \gamma\lambda\epsilon_{t-1}(s,a) + [s = s_t, a = a_t],
\end{aligned}$$

только теперь еще надо забывать следы, если мы не следуем стратегии: $\epsilon_t(s,a) := 0$, если $Q(s_t,a_t) \neq \max_a Q(s_t,a)$.

О классическом обучении с подкреплением можно говорить еще долго, но в этой главе нам будет достаточно основных принципов TD- и Q-обучения. Напоследок же — небольшое замечание по поводу источников для интересующихся.

[1] Догадайтесь, почему алгоритм называется SARSA.

В обучении с подкреплением сложилась ситуация, совершенно уникальная для информатики: лучшей книгой по обучению с подкреплением до сих пор остается книга Ричарда Саттона и Эндрю Барто [519], изданная еще в 1998 году[1]! Тем, кто хочет подробнее разобраться в обучении с подкреплением, рекомендуем просто прочесть эту книгу от начала до конца (она доступна свободно), она небольшая и несложная, но в ней очень ясно и подробно изложены и все основные идеи этого раздела, и многие их расширения.

9.3. От TDGammon к DQN

> Его взор, прежде взор простого наблюдателя, впивается теперь глубже, сумрачнее, упорнее, неотступнее...
>
> *С. Цвейг. Зигмунд Фрейд*

Все те алгоритмы обучения с подкреплением, о которых мы до сих пор говорили, использовали значения вида $Q(s,a)$ или $V(s)$, причем они пытались получить эти значения в явном виде, например, обучить функцию Q^* на всех возможных входах (s,a), чтобы потом максимизировать ожидаемый результат, найдя оптимальную стратегию. Но ведь этих самых состояний s обычно астрономическое число — представьте, сколько возможных позиций в игре го! А если число состояний умножить на число возможных действий в них, получится еще больше. Поэтому в реальности, конечно, никто не пытается перечислить все возможные состояния, построив и обучив огромную таблицу $Q(s,a)$. Подход обычно такой:

- давайте представим входы, то есть состояния $s \in S$ и действия $a \in A$, в виде каких-то характерных признаков, так, чтобы размерность входа перестала быть астрономической;
- а функцию $Q(s,a)$, в которой раньше значения на разных входах были независимы друг от друга, представим как какую-то параметрическую модель машинного обучения $Q(s,a;\boldsymbol{\theta})$, на вход которой подаются признаки, описывающие s и a;
- тогда функция $Q(s,a;\boldsymbol{\theta})$ — это просто сложная функция из признаков в одно вещественное число (ожидаемый выигрыш, или его вероятность в случае

[1] Небольшое лирическое отступление: в информатике действительно обычно свежие книги лучше старых просто потому, что информатика развивается очень быстро, и новые книги передают новые результаты, которые часто поглощают старые, дают свежий взгляд и более глубокое понимание. Но в математике в целом ситуация зачастую обратная. Мы неоднократно убеждались, что, например, лучшие учебники по математическому анализу и дифференциальным уравнениям писали в начале и середине XX века, с расчетом на тогдашних инженеров; в этих книгах очень хорошо дается именно понимание происходящего, упор делается на связь математики с реальным миром и решение содержательных задач, а не только механическое переписывание формул. Так что если у вас на антресолях пылятся старые книги по математике, откройте их, возможно, будете приятно удивлены.

бинарного исхода), и ее параметры $\boldsymbol{\theta}$ можно пытаться обучать методами машинного обучения;

- входами для обучения будет, согласно идее TD-обучения, каждый очередной переход $(s_t, a_t, r_{t+1}, s_{t+1})$;
- и каждый шаг обучения выглядит так: агент делает ход a из состояния s, переходит в новое состояние s', получая за это непосредственную награду r, а затем делает один шаг обучения функции $Q(s,a;\boldsymbol{\theta})$ со входом (s,a) и выходом $\max_{a'} Q(s',a';\boldsymbol{\theta})+r$ (напомним, что в подавляющем большинстве случаев $r = 0$, награду обычно дают только в самом конце эпизода обучения); кроме того, агент может также сделать такие шаги по отношению к предыдущим позициям, обновив веса не только для последнего входа, но и для нескольких предыдущих.

В этой схеме нейронные сети, как обычно, отлично работают в качестве универсального черного ящика, который может приблизить любую функцию, в том числе и функцию $Q(s,a)$. Например, если под обучением понимать обычный градиентный спуск, то мы обновляем веса нейронной сети по следующему правилу:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha\,(y_{t+1} - y_t)\sum_{k=1}^{t}\lambda^{t-k}\nabla_{\boldsymbol{w}} y_k,$$

где y_k — выход нейронной сети на шаге k, а λ — это параметр затухания, который показывает, насколько сильно нужно учитывать результаты последнего хода для пред-предыдущего, пред-пред-предыдущего и так далее. Получается, что при $\lambda = 0$ мы обновляем веса только на основании последнего хода, а при $\lambda = 1$ рассматриваем всю историю от начала времен (то есть от начала эпизода обучения, одной «партии»).

Первые успехи нейронных сетей на поприще обучения с подкреплением относятся к временам весьма древним. В 1992 году Джеральд Тезауро (Gerald Tesauro) разработал программу, получившую название TD-Gammon [525, 526]; название вполне логичное, ведь играла эта программа в нарды (backgammon), и делала это при помощи TD-обучения. TD-Gammon работает в точности как написано выше, используя обычную неглубокую нейронную сеть с одним скрытым слоем для обучения по правилу выше. Нарды оказались очень благодатной почвой для такого подхода, потому что ход игры зависит от бросков кубика, а это значит, что можно исследовать значительную часть пространства поиска, просто играя с самим собой, кубики позаботятся о всех необходимых случайностях. И это именно то, что делала TD-Gammon для обучения: просто играла сама с собой миллионы партий, постепенно обучаясь все лучше и лучше; волшебство здесь в том, что никакого обучающего набора оказывается не нужно.

TD-Gammon ждал оглушительный успех. С точки зрения обучения добрым знаком было то, что модель хорошо масштабировалась: при увеличении размера сети и времени, выделенном на обучение, сеть начинала играть все лучше. По мере

обучения TD-Gammon сначала обучалась простейшим элементам стратегии и тактики нард, а потом постепенно начинала выделять более сложные признаки. В результате программа даже на простом представлении позиции без всяких хитростей начинала играть очень сильно, а при добавлении к представлению нескольких вручную порожденных признаков из более ранней программы Neurogammon [524] TD-Gammon начинала успешно соперничать с людьми-чемпионами.

Единственными слабостями TD-Gammon были ошибки в использовании удвоения ставок и плохая игра в эндшпиле: TD-Gammon смотрела только на два хода вперед, а эндшпили даже в нардах требуют более глубокого расчета. Поэтому в выставочных играх TD-Gammon немного уступила тогдашним чемпионам. Тем не менее, TD-Gammon оказала серьезное влияние на развитие нард: некоторые классические позиции были полностью переоценены игроками-людьми, потому что оценка TD-Gammon противоречила человеческой интуиции и практике игр; тот же эффект можно было наблюдать позднее и в шахматах, а теперь и в го.

Однако TD-Gammon надолго осталась практически единственной успешной программой, основанной на идеях обучения нейронной сети с подкреплением. Разумеется, исследователи тут же попытались применить аналогичных подход к шахматам и го, но у них мало что получилось. Последовали даже относительно пессимистические работы, которые показывали, что Q-обучение с нелинейными параметрическими приближениями (а нейронная сеть — это самое нелинейное приближение, которое только бывает) часто расходится [541], а успех TD-Gammon объяснялся исключительно упомянутым выше эффектом равномерного распределения обучения по пространству поиска за счет кубиков [428].

К счастью, эти пессимистические прогнозы не оправдались; по мере того, как развивалась революция глубокого обучения, начали появляться и попытки моделировать функции V и Q, а также окружающую агента среду, при помощи глубоких сетей. После ранних работ [210, 469] прорыв, окончательно определивший направление для современных успехов, был достигнут в работе Володимира Мниха (Volodymyr Mnih) с соавторами из компании Google DeepMind, в которой они применили идеи обучения с подкреплением к ранним, но от этого не менее привлекательным играм для приставок и аркадных автоматов Atari; да-да, тем самым *Pong*, *Space Invaders*, *Breakout* и другим классическим хитам 1980-х годов.

Первая работа была выложена на arXiv в 2013 году [426], а немного улучшенная и примененная к большему числу разных игр модель была описана в статье 2015 года, вышедшей в одном из главных научных журналов мира, *Nature* [238][1]. Этот подход получил название *глубокого обучения с подкреплением* (deep reinforcement learning), а сети, обученные таким способом, называются *глубокими Q-сетями* (deep Q-networks, DQN).

[1] Кстати, статья про AlphaGo тоже появилась в Nature [355]; результаты, безусловно, блестящие, но сама идея того, что статьи о компьютерных программых, играющих в настольные и компьютерные игры, печатают в Nature, для нас всегда была немного контринтуитивной.

У DQN в варианте [238] есть некоторые небольшие, но важные усовершенствования по отношению к базовой архитектуре, описанной выше. Во-первых, практика показывает, что обучаться непосредственно на последовательных кадрах игры — плохая идея: соседние кадры слишком похожи друг на друга, сильно коррелируют, причем со временем их распределение, естественно, сдвигается в зависимости от хода игры, но остается локализованным.

Это мешает эффективному обучению, ведь в обычной постановке задачи обучения мы предполагаем, что тренировочные данные независимы, а распределение данных со временем не меняется. Поэтому по мере обучения DQN сначала накапливает некоторый опыт, сохраняя свои действия и их результаты на протяжении какого-то времени, а потом выбирает из этого опыта случайный мини-батч отдельных примеров для обучения, взятых в случайном порядке; для накопления опыта при обучении использовалась ϵ-жадная стратегия. Формально говоря, на каждом шаге обучения t мы:

- выбираем следущее действие a_t (в ϵ-жадной стратегии мы выбираем случайное действие с вероятностью ϵ и $a_t = \arg\max Q(s_t, a; \boldsymbol{\theta})$ в противном случае;
- делаем это действие, получая награду r_t и следующее состояние s_{t+1}; новая единица опыта (s_t,a_t,r_t,s_{t+1}) записывается в память;
- затем выбираем из памяти случайный мини-батч таких «единиц опыта» для обучения; для простоты пусть это будет одна единица (s_j,a_j,r_j,s_{j+1});
- подсчитываем выход сети y_j (об этом чуть ниже) и делаем один шаг градиентного спуска для функции ошибки $L = \big(y_j - Q(s_j,a_j;\boldsymbol{\theta})\big)^2$; это значит, что мы сдвигаем веса сети на

$$\nabla_{\boldsymbol{\theta}} L = 2\left(y_j - Q(s_j,a_j;\boldsymbol{\theta})\right)\nabla_{\boldsymbol{\theta}} Q(s,a;\boldsymbol{\theta}).$$

Во-вторых, важную роль для успеха сыграло то, что при обучении DQN сеть, которая отвечала за целевую функцию (target network), была отделена от сети, которая собственно обучается. Практика показывает, что если применять TD-обучение напрямую, слишком мощные аппроксиматоры (например, нейронные сети) обнаруживают несомненные склонности к галлюцинациям: они быстро заходят в какие-то ими самими придуманные локальные экстремумы и начинают очень глубоко исследовать совершенно бессмысленные части пространства поиска, бесцельно тратя ресурсы и фактически не обучаясь.

К счастью, этой беде относительно легко помочь: достаточно сделать так, чтобы сеть не сразу использовала обновленную версию в целевой функции, а обучалась достаточно долгое время по старым образцам, прежде чем сделать уже полноценный глобальный апдейт. Иными словами, в приведенном выше алгоритме мы считаем

$$y_j = \begin{cases} r_j, & \text{если эпизод обучения закончился,} \\ r_j + \gamma \max_{a'} Q(s_j,a_j;\boldsymbol{\theta}_0), & \text{если нет,} \end{cases}$$

где $\boldsymbol{\theta}_0$ — это некие зафиксированные веса, которые не меняются после каждого тестового примера, меняются только $\boldsymbol{\theta}$. В какой-то момент, обычно один раз за один или несколько эпизодов обучения, нужно возвращаться к этим весам целевой функции и присваивать $\boldsymbol{\theta}_0 := \boldsymbol{\theta}$.

Де-факто у нас параллельно обучаются две сети: одна определяет поведение, а другая — целевую функцию; структура у них одна и та же, и обучаются они одинаково на одних и тех же данных, но одна сеть постепенно отстает от другой, время от времени скачком «догоняя» первую.[1]

В-третьих, стоит отметить, что на практике обычно применяется архитектура, в которой возможное действие агента $a \in A$ не подается на вход, а просто у сети столько выходов, сколько возможных действий, и на входе $s \in S$ сеть пытается предсказать результаты каждого действия (после чего, естественно, выбирает максимальный). Это важное улучшение, потому что оно позволяет получить ответы для сразу всех действий за один проход по сети, что ускоряет происходящее в разы, а сеть от этого сильно сложнее не становится, ведь основная часть ее «логики» остается прежней и используется заново.

И наконец, еще одно важное улучшение состоит в переходе к так называемой *dueling network*: мы разделяем Q-сеть на два канала, один из которых вычисляет оценку позиции $V(s;\boldsymbol{\theta})$, которая является функцией позиции и не зависит от текущего действия a, а другой — зависящую от действия *advantage function* (функцию преимущества) $A(s,a;\boldsymbol{\theta})$. На последнем этапе они просто складываются:

$$Q(s,a;\boldsymbol{\theta}) = V(s;\boldsymbol{\theta}) + A(s,a;\boldsymbol{\theta}).$$

Таким образом, одна часть сети обучается оценивать позицию как таковую, а другая — предсказывать, насколько полезны будут разные наши действия в этой позиции. Кстати, обратите внимание, что мы, конечно же, не можем отличить одно от другого в обучающей выборке, это всего лишь слегка измененная архитектура сети, а обучаем мы по-прежнему функцию $Q(s,a;\boldsymbol{\theta})$, но это изменение в архитектуре дает нужный «намек» и часто существенно улучшает результаты.

В работах [238, 426] этот подход применили к тому, чтобы играть в игры Atari, причем входом служило не состояние игры (это потребовало бы отдельных инженерных решений для каждой игры, а именно этого и хотелось бы избежать), а собственно игровое поле в виде картинки из пикселей, той самой, которую видит на экране игрок-человек. Точнее говоря, на вход подавалась конкатенация последних четырех кадров игры, то есть можно сказать, что долгой памятью агента не снабдили, он едва-едва мог оценить скорости разных объектов на экране.

---

[1] В алгоритмических разделах информатики в подобных ситуациях часто используются звериные метафоры; например, в «алгоритме зайца и черепахи» при поиске цикла в графе указатель-заяц постепенно обгоняет указатель-черепаху «на круг», и мы ждем, когда они снова встретятся. А здесь у нас скорее «алгоритм зайца и кенгуру»: сеть, определяющая поведение, обучается быстро и понемногу, маленькими шажками, а сеть, определяющая целевую функцию, время от времени величавым прыжком нагоняет зайца в очередном чекпойнте.

Единственное послабление, которое сделали агенту, состояло в том, что его не заставляли учить читать цифры очков[1], а просто давали число очков из игры в явном виде (замазывая его при этом на картинках); в итоге пришли даже к еще более простой схеме: выдавали награду +1 за каждое позитивное достижение в игре и −1 за каждое негативное событие (в Atari это обычно потеря очередной «жизни»).

В результате получилось, что такая сеть, ничего не зная собственно о «правилах игры», просто по входной картинке и целевой функции обучилась играть во многие игры Atari лучше человека. Любопытно, однако, что не во все. Игры без долгосрочной стратегии вроде *Breakout* или *Video Pinball* покорились DQN без вопросов, результаты были вдесятеро выше человеческих (напомним, что речь идет о людях–экспертах, лучших игроках мира в соответствующей игре). Результаты на уровне человеческих или немного лучше получились в играх вроде *Pong* и *Space Invaders*, где стратегия есть, но она не очень обязательна и не слишком долгосрочна. А вот в играх, где нужно думать надолго вперед, ничего не получилось; например, в *Montezuma's Revenge* DQN играть совершенно не обучилась, устойчиво получая нулевые результаты. Похожий подход привел к успеху и в игре го, но здесь, чтобы та самая «долгосрочная стратегия» все же сработала, добавятся еще несколько важных идей, поэтому об AlphaGo мы поговорим отдельно в разделе 9.4.

И еще одно замечание: обучение с подкреплением для игр и других подобных отличается от большинства других задач машинного обучения тем, что здесь у нас есть фактически *неограниченный* источник новых тренировочных примеров. В случае игр *Atari* мы можем запускать симулятор игры сколько угодно раз, пробуя самые разные стратегии и обучая модель хорошо играть. В случае игры в нарды или го модель может сколько угодно играть сама с собой, тоже получая практически неограниченную последовательность примеров для обучения. И хотя примеры эти будут в некотором смысле зависеть от текущей версии модели — в конце концов, именно она (обычно с добавленным случайным шумом для исследования) играет, когда создаются новые тренировочные примеры, — это все равно не идет ни в какое сравнение с обычными ситуациями, когда есть некий фиксированный датасет, и максимум, что вы можете сделать, — добавить в него немного случайного шума, как в шумоподавляющем автокодировщике (см. раздел 5.5). С другой стороны, обучение хорошей стратегии методами обучения с подкреплением в любой достаточно сложной ситуации — это процесс долгий и сложный: те самые модели DQN для игр *Atari* даже на самых современных видеокартах обучаются по нескольку дней, прежде чем могут показать сколь-нибудь разумные результаты.

Поэтому вполне логично, что в глубоком обучении с подкреплением основной упор дальнейших исследований пока что был сделан не на максимально эффективное использование каждого тренировочного примера (их легко нагенерировать еще), а на том, чтобы максимально быстро обучаться.

---

[1] Да и с чего бы агент взял, что эти цифры нужно увеличивать и что на них вообще есть какой-то порядок... Вообще, целеполагание — это еще слабое место искусственного интеллекта. У нейронных сетей пока явные проблемы с мотивацией: может быть, искусственный интеллект уже давно поработил бы мир, да как-то не хочется...

Следующий прорыв в скорости обучения был сделан на почве *асинхронного* обучения с подкреплением, использующего две особенности обучения DQN:

- во-первых, обучение происходит не после каждого хода, а путем случайного сэмплирования из накопленной памяти, и для обучения нужно сначала собрать некоторое количество тестовых примеров, а только потом обновлять веса модели;
- во-вторых, сеть, которая генерирует целевые значения для функции потерь, — это не та же самая сеть, которая обучается после каждого мини-батча, она может и даже должна существенно отставать от сети, генерирующей поведение агента.

Поэтому оказалось, что обучение с подкреплением можно разбить на несколько практически независимых частей, которые должны делиться друг с другом новостями только в определенные достаточно далеко друг от друга отстоящие моменты времени, а между ними могут работать совершенно параллельно и независимо:

- должен все-таки быть некий центральный процесс, сервер, который хранит текущие значения параметров, обновляет их по мере надобности и раздает всем остальным;
- первый вид процессов — это собственно «игроки», которые взаимодействуют с окружающим миром и нарабатывают новый опыт; им нужно время от времени получать от сервера обновления параметров модели (они используются при выборе действий), а сами они просто накапливают единицы опыта в виде тех самых четверок (s_t,a_t,r_t,s_{t+1}) и передают накопленный опыт в общее хранилище памяти;
- второй вид процессов, «обучатели», получают из хранилища памяти опыт в виде мини-батчей единиц опыта и считают градиенты функции ошибки; им нужна для этого сеть, генерирующая целевые значения, и текущая, так что «обучатели» находятся в более тесном контакте с сервером; но заметим, что они по-прежнему совершенно независимы, каждый из них считает свое собственное значение градиента и свои собственные обновления для весов модели;
- наконец, собственно сервер собирает все эти обновления, применяет их к хранящейся у него модели, и в какой-то момент (обычно регулярный, но достаточно редкий) раздает обновленную модель обратно «игрокам» и «обучателям», а также обновляет модель, генерирующую целевые значения; получается, что синхронизация в такой архитектуре, конечно, нужна, но ее можно делать относительно редко.

В работе [354] этот подход был применен для того, чтобы распараллелить обучение с подкреплением на кластер из нескольких компьютеров: в разработанной авторами архитектуре с характерным названием *Gorila* (от слов General Reinforcement Learning Architecture) каждый из процессов, описанных выше, может быть реализован на отдельном компьютере.

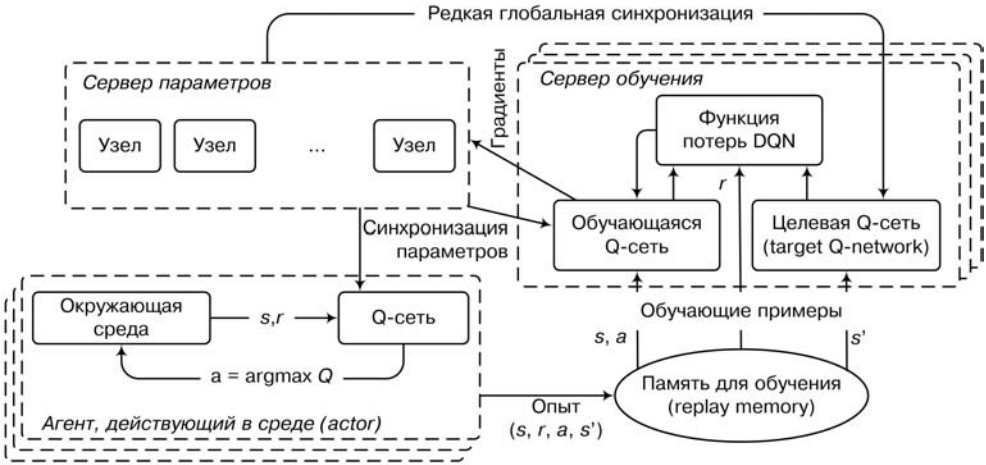


Рис. 9.4. Распределенная архитектура Gorila [354]

А потом выяснилось, что можно распараллелить все это еще эффективнее, если в качестве отдельных агентов использовать потоки одного и того же процессора; тогда они могут просто обращаться к одной и той же модели в памяти, но при этом все равно делать свое дело относительно независимо.

Мы изобразили архитектуру Gorila на рис. 9.4. Обратите внимание, что параллелизовано здесь буквально *все*. Во-первых, несколько независимых «игроков» (actors) действуют каждый в своей копии среды, порождая новые единицы опыта и передавая их в глобальную память (replay memory). Во-вторых, «обучателей» (learners) тоже несколько: каждый содержит копию Q-сети и вычисляет градиенты по своему очередному мини-батчу тренировочных примеров, взятому из памяти; для подсчета градиентов используется целевая Q-сеть (target network). В-третьих, градиенты эти отправляются на сервер параметров, который тоже содержит несколько параллельных узлов (shards), каждый из которых хранит и обновляет свою отдельную часть вектора параметров сети $\boldsymbol{\theta}$.

Любопытно, что асинхронное обучение оказывается не просто быстрее, но и существенно лучше. Причины такого эффекта точно не известны, но, по всей видимости, дело в том же эффекте, который помогает стохастическому градиентному спуску: распределенное асинхронное обучение приводит к тому, что модель не пытается слишком детально исследовать одну и ту же часть пространства поиска. Разные «игроки» ведут себя по-разному, и их «предвзятые» отношения к различным частям пространства поиска усредняются. В работе [11] первые результаты DQN на играх Atari были превышены в разы при том, что общее время обучения сократилось; любопытно, что для обучения при этом не использовались видеокарты, и все делалось в 16 потоков на обычном современном процессоре.

9.4. Бамбуковая хлопушка

> Рэндзю — удел простолюдинов, в шахматы играют герои, го — игра богов.
>
> *Японская пословица*

> Мэйдзину уже не раз приходилось вести матчи, где на карту ставилась его судьба, и в такого рода матчах он не проиграл ни разу. Его игра была мощной и до того, как он стал Мэйдзином, а игры, которые он вел, уже получив титул, уверили весь мир в его непобедимости. То, что он сам в нее верил, мало того, хотел верить, — лишь усугубляло трагедию.
>
> *Я. Кавабата. Мэйдзин*

29 января 2016 года. Google торжественно объявляет о том, что достигнута важна договоренность: через месяц состоится матч между Ли Седолем (Lee Sedol), национальным героем Кореи, великим чемпионом по игре го, который и сейчас остается одним из лучших игроков мира, и недавно появившейся программой AlphaGo, разработанной в Google DeepMind и несколько месяцев назад победившей чемпиона Европы Фань Хуэя[1].

Кажется, что для Ли Седоля появление программы, за победу в матче с которой Google предлагает миллион долларов, — всего лишь способ заработать: он абсолютно уверен в себе и не допускает даже мысли о поражении. На пресс-конференции Ли Седоль отдает должное разработчикам программы и признает, что AlphaGo, возможно, представляет собой новый шаг в развитии компьютерного го, но о результате матча говорит однозначно: «В этот раз я, конечно, выиграю со счетом 4:1 или 5:0, но я уверен, что через 2–3 года создатели AlphaGo захотят взять у меня реванш. Вот тогда мне будет действительно непросто победить» [479].

8 марта 2016 года. На пресс-конференции накануне матча Ли Седоль по-прежнему не сомневается в том, что он сильнее компьютера, но заявления чемпиона становятся более осторожными: «Теперь мне кажется, что AlphaGo может до некоторой степени имитировать человеческую интуицию. Мне, наверное, стоит немножко волноваться за исход матча... Но я все равно уверен в своей победе».

12 марта 2016 года. Ли Седоль пожимает руку Адже Хуаню (Aja Huang), делавшему ходы за AlphaGo, и сдается в третьей подряд партии матча. Счет становится 3:0 в пользу AlphaGo; Ли Седоль сможет в блестящем стиле победить

---

[1] Целомудренная транскрипция; а сам факт того, что переехавший в Европу из Китая второй профессиональный дан Фань Хуэй стал чемпионом Европы, к сожалению, неплохо характеризовал уровень европейского го. Но сейчас уже намечается несомненный прогресс, и здесь Россия идет впереди остальной Европы: в России появились сразу несколько игроков профессионального уровня, а в 2020 году во Владивостоке пройдет чемпионат мира среди любителей — до этого этот турнир почти всегда проходил в Японии, несколько раз в Китае и буквально по одному разу в Корее и Таиланде.

в четвертой партии, но через три дня матч закончится со счетом 4:1, а сама программа AlphaGo получила почетный сертификат девятого профессионального дана как выполнившая требования профессиональной федерации. Подводя итоги матча, Ли Седоль написал: «Неделя с AlphaGo была для меня как бамбуковая хлопушка[1]. Слабости "го Ли Седоля" обнажились полностью. Старые, косные взгляды профессионального сообщества го разбились вдребезги. Нам нужно снова обдумать и пересмотреть ходы, которые мы принимали как должное» [478].

Скажете, это история безмерной гордыни все еще молодого корейского профессионала? Ведь, казалось бы, давно понятно, что с компьютерами в настольных играх, тем более в играх с полной информацией, шутки плохи: Гарри Каспаров проиграл DeepBlue еще за двадцать лет до описываемых событий, а современные шахматные программы играют на полтысячи очков Эло сильнее даже самых лучших гроссмейстеров. Неужели Ли Седоль был настолько неосведомлен о вычислительной мощи современных компьютеров?

Вовсе нет. AlphaGo действительно стала сенсацией; практически ни один эксперт не предсказывал перед матчем ни столь уверенной победы программы, ни ее победы вообще[2]. В этой главе мы поговорим о том, почему компьютерное го — это так сложно, опишем, как устроена AlphaGo и что она добавила к уже существовавшим программам для игры в го, подробнее рассмотрим собственно глубокие модели, использующиеся при этом, и обсудим еще несколько интересных применений глубокого обучения с подкреплением. Но сначала нам нужно поговорить о том, что это, собственно, такое.

Спустя год после матча с Ли Седолем, на специально собранном конгрессе *Future of Go Summit*, прошедшем в конце мая 2017 года в Вужене (Китай), состоялся матч между AlphaGo и номером 1 мирового рейтинга, китайцем Кэ Цзе (Ke Jie). Любопытно, что против Кэ Цзэ играла так называемая версия AlphaGo Master, которая обучалась больше на играх с самой собой, чем на играх профессионалов го, и работала всего лишь с одного компьютера на Google Cloud с TPU, а не на распределенной сети из 1920 CPU и 280 GPU, как игравшая с Ли Седолем программа. Тем не менее, больших проблем у AlphaGo не возникло: хотя первая партия была очень напряженной, и оценка позиции самой AlphaGo очень долго оставалась равной, в итоге Кэ Цзе был повержен со счетом 3:0. Кроме того, AlphaGo выиграла партию против команды из пяти ведущих профессионалов.

Что же произошло? Как устроена AlphaGo и почему ей удалось обыграть человека в самую «человеческую» из дискретных игр с полной информацией?

---

[1] Бамбуковые хлопушки используются в практике дзен-медитации: в большую хлопушку время от времени бьют для того, чтобы медитирующие не засыпали; отсюда и образ, использованный Ли Седолем — AlphaGo «открыла ему глаза».

[2] Прогнозы российских любителей и профессионалов можно прочесть в журнале российской гофедерации «оГо»: http://rusgo.org/magazine13/. Один из авторов книги тоже принял участие в опросе, и явно переосторожничал, предсказав, что Ли Седоль может поначалу недооценить AlphaGo и проиграть одну партию...

Сначала — немного истории. Первые программы для игры в го, как и для шахмат, появились еще в конце шестидесятых; самая известная из них связана с именем Альберта Зобриста. Однако быстро стало понятно, что научиться хорошо играть пока невозможно, и про го на некоторое время забыли. Вернулся интерес в середине восьмидесятых, когда у многих появились персональные компьютеры, а также появились шахматные программы, играющие достаточно сильно (но пока еще не обыгрывающие чемпионов). С 1987 года чемпионат по го среди компьютерных программ проводился под эгидой Фонда Ина (Ин Чанци, Ing Changqi — тайваньский мультимиллионер, любитель и популяризатор игры го); фонд учредил и премию в 40 миллионов тайваньских долларов (около полутора миллионов американских) для создателей программы, которая смогла бы обыграть одного из китайских или тайваньских профессионалов игры го (любой силы, по сути нужно было обыграть игрока первого профессионального дана). Приз поддерживался до 2000 года, в девяностые появились такие известные и важные программы, как Handtalk и Many Faces of Go... и как же они играли?

В 1997 году *Deep Blue* обыграл в официальном матче Гарри Каспарова. В том же 1997 году программа Handtalk получила часть приза фонда Ина, победив со счетом 2:1 трех любителей игры го уровнем от второго до шестого дана[1]. Звучит не так плохо, это примерно аналогично кандидату в мастера спорта по шахматам... вот только любителям этим было от 11 до 13 лет, а играли они *на 11 камнях форы*. В го есть очень логичная и удобная система гандикапа, которая часто применяется в турнирах, но в партиях между людьми больше девяти камней форы практически никогда не встречаются; обыграть любителя среднего дана на 11 камнях форы — это примерно как обыграть кандидата в мастера по шахматам, который дал вам ферзя и ладью вперед.

Почему же го — это так сложно? Во многих других играх (например, в шахматах) отлично работает так называемый *поиск с альфа-бета-отсечением* (alpha-beta search): мы строим минимакс-дерево ходов, выбрасываем ходы, которые заведомо хуже других, тем самым обрезая бесперспективные ветви дерева, и ведем поиск в глубину. Примерно так работал и *Deep Blue*. Но го — это совсем другое дело: в обычной шахматной позиции 30–40 возможных ходов, из которых многие сразу приводят к значительным материальным потерям, и их можно тут же перестать рассматривать, а в го около 200 возможных ходов (любое поле на доске 19×19, кроме занятых), и из них «вполне разумных» тоже около сотни. С оценкой позиции тоже сложно: в шахматах можно посчитать материал и какие-то простые позиционные эвристики, и если разрыв большой, то оценка позиции, скорее всего, очевидна; в го тоже может упасть большая группа, но это редкость, обычно потери будут территориальными и очень трудными для оценки. Кстати, вот вам очень сложная для

[1] В го есть существенная разница между профессионалами (это официальный титул) и любителями — любительские даны считаются (и являются) значительно слабее профессиональных, то есть сила эти игроков не достигала требуемого уровня первого профессионального дана.

автоматизации задача: по данной *финальной* позиции в го, когда люди уже закончили игру, определить, кто же, собственно, выиграл. С тактикой, где компьютеры должны, по идее, «обсчитывать» людей, тоже в го не все так просто: тактика там всегда завязана на стратегию и взаимодействие камней по всей доске, а кроме того, человеку в го считать значительно проще, чем в шахматах (потому что «фигуры» почти никогда не двигаются). И это мы еще не говорим о ко-борьбе[1], которую программы всегда вели просто отвратительно.

Первая революция в компьютерном го произошла в 2006 году, когда для этой игры начали использовать *поиск по дереву Монте-Карло* (Monte-Carlo tree search, MCTS) [190, 515]. Впервые реализованный в MoGo, а затем и во всех других программах, MCTS работает на удивление просто: чтобы оценить позицию, мы запускаем *случайные симуляции*, начинающиеся с этой позиции, смотрим, в каких ветках черные или белые выиграли больше *случайных* партий, а затем рекурсивно повторяем этот поиск в самых перспективных узлах получающегося дерева. Основная часть MCTS – формула, по которой выбирают узел для дальнейшего анализа. Все это основано на тех же многоруких бандитах, о которых мы говорили в разделе 9.1, и алгоритм UCT (upper confidence bound UCB1 applied to trees) выбирает узел с максимальным приоритетом

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}},$$

где w_i — число побед в узле i, n_i — число симуляций в нем, c — параметр, часто $\sqrt{2}$, а $t = \sum_i n_i$ — общее число симуляций; сравните с UCB1.[2]

Идея доигрывать позицию до конца случайными ходами выглядит очень странно, но тут же пошли более разумные результаты: в 2008 году MoGo достиг уровня дана на доске 9×9, а в 2009 году Fuego победила одного из топ-игроков на доске 9×9, но с «настоящим» го, на доске 19×19, все шло медленнее: в 2009 году Zen достиг уровня первого любительского дана, в 2012 году новая версия Zen выиграла матч у любителя второго дана на полноразмерной доске со счетом 3:1, а в 2013 году CrazyStone выиграл у профессионала девятого дана (наивысший ранг в го), получив четыре камня форы. Но весь этот прогресс был довольно медленным и постепенным, в игре компьютерных программ было много очевидных пробелов, и ничто не предвещало внезапный успех AlphaGo.

---

[1] Ко-борьба — это специальный раздел тактики го, который происходит из запрета на повторение позиции. Мы не будем вдаваться в детали, хотя всецело рекомендуем читателям попробовать го на практике: правил там очень мало (куда меньше, чем в шахматах), но из них происходит одна из самых глубоких и интересных существующих игр.

[2] Идея алгоритма MCTS, конечно, применима не только к го. Упомянем самый общий вариант, General Game Playing. Это соревнование между программами, которым на вход подаются правила новой, ранее не известной им игры (естественно, не текстом, а в унифицированном формате), и они должны тут же начать в нее играть и выигрывать. В 2007 году программа CadiaPlayer, основанная на MCTS, стала чемпионом по General Game Playing, и с тех пор этот подход стал общепринятым [159, 160].

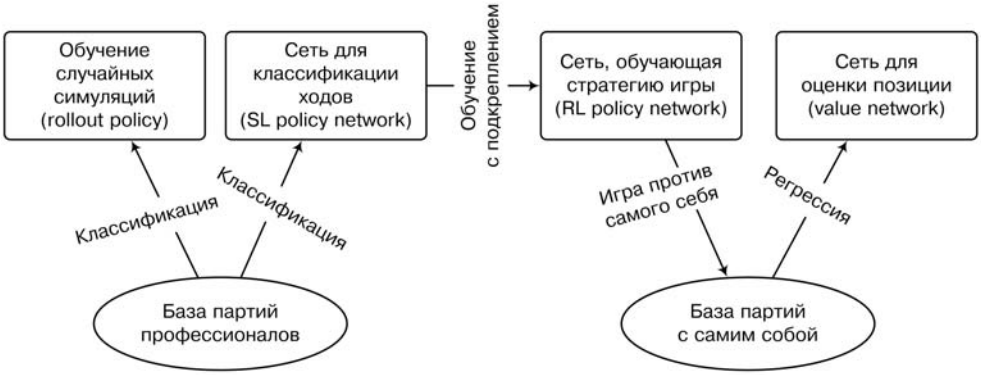


Рис. 9.5. Архитектура AlphaGo [355]

Давайте более подробно посмотрим на то, как именно работает AlphaGo [355]; общая ее архитектура изображена на рис. 9.5. В AlphaGo несколько сетей, которые обучались последовательно. Сначала SL policy network (от слов supervised learning) обучалась предсказывать ходы: 13-слойная сверточная сеть выделяет признаки по всей доске и обучается на профессиональных играх (30 миллионов позиций с сервера KGS), на выходе у сети получается распределение вероятных ходов $p_\sigma(a \mid s)$.

Сеть SL policy network в составе AlphaGo была способна правильно предсказать 57,0 % ходов профессионалов — это очень, очень много, если учесть, что в большинстве позиций вполне разумных ходов может быть несколько. Таким образом, после этого у нас есть сеть, которая может неплохо предсказывать ходы. Кроме того, вдобавок мы обучаем быструю, но не такую умную стратегию p_π: точность у нее «всего» 24,2 %, но одну позицию она способна оценить за две наносекунды (!) вместо трех миллисекунд.

Вторая сеть — RL policy network, от слов reinforcement learning. Это уже сеть, реализующая стратегию обучения с подкреплением, и работает она так: все та же базовая структура 13-слойной сверточной сети инициализируется весами SL policy network, а затем играет сама с собой (точнее, с одной из предыдущих итераций обучения). Веса при этом дообучаются так, чтобы максимизировать вероятность выигрыша. На этом этапе, после (долгого, очень долгого) обучения с подкреплением получается RL policy network, которая самостоятельно, безо всякого расчета вариантов, просто глобальным приближением к оптимальной стратегии выигрывает 80 % игр против SL policy network и около 85 % игр против *Pachi*, одной из лучших MCTS-программ!

Затем нужно использовать полученную стратегию, чтобы обучить собственно функцию оценки позиции $V_\theta(s)$. Мы будем предсказывать результат игры по позиции нейронной сетью (value network), структура которой опять похожа на policy

network, но теперь имеет только один выход (предсказание результата). Если пытаться обучать ее на наборе профессиональных игр, мы довольно быстро ударимся в оверфиттинг, данных в профессиональных партиях явно недостаточно. Поэтому $V_\theta(s)$ обучается на наборе игр RL policy network против самой себя; таких мы можем сгенерировать сколько угодно. В результате получается сеть для глобальной оценки позиции, которая работает очень быстро по сравнению с MCTS и дает качество на уровне MCTS с RL policy (но в 15 тысяч раз быстрее) и гораздо лучше, чем обычный MCTS.

Ну а теперь, когда готова функция оценки позиции, можно и деревья посчитать. AlphaGo строит MCTS-подобное дерево, в котором априорные вероятности инициализируются через SL policy network как $p_\sigma(a \mid s)$. В каждом листе L AlphaGo подсчитывает значение value network $V_\theta(s_L)$ и результат случайных доигрываний этой позиции z_L, порожденных при помощи стратегии p_π, а потом объединяет полученные результаты. Любопытно, что в экспериментах AlphaGo SL policy network для построения дерева сработала лучше (видимо, потому, что она дает более разнообразные варианты), но функцию оценки позиции лучше строить на основе более сильно играющей RL policy network.

Вот так и получилась «бамбуковая хлопушка» для Ли Седоля и одно из самых ярких событий в мире искусственного интеллекта в последние годы. А летом 2016 года на европейском го-конгрессе в Петербурге чемпион Европы Фань Хуэй сделал доклад о том, как AlphaGo может помочь развитию самой игры. Во-первых, успех AlphaGo стал огромным имиджевым успехом для го как игры; во время недели матча Ли Седоля с AlphaGo в мире продавалось в десять раз больше гобанов (досок для игры), чем обычно. Что же до сути игры, Фань Хуэй говорил о том, что AlphaGo стала первой программой, которая действительно следовала важнейшей заповеди го, сформулированной легендарным китайским игроком Сюсаку Хонинбо: «Жадность никогда не побеждает». Многие великие игроки отмечали, что в го очень важно играть спокойно, оставив эмоции; таким спокойным стилем известен никогда не улыбающийся великий чемпион недавнего прошлого, кореец Ли Чанг-Хо; и в этом тоже, конечно, компьютерные программы превосходят людей.

AlphaGo, по словам Фань Хуэя, может дать новое видение мировому го, предложить новые ходы, которые даже лучшие игроки не всегда могут увидеть: в описанных выше пяти партиях с Ли Седолем было несколько таких примеров. Наконец, Фань Хуэй впервые показал визуализацию того, как AlphaGo на самом деле выбирает ходы, и визуализацию оценки позиции, которую делает AlphaGo в процессе игры. Он показал несколько конкретных примеров того, как AlphaGo меняет классическую оценку стандартных позиций, отклоняется от проверенных десятилетиями дзесеки (стандартных продолжений) и получает позиции не хуже, а скорее лучше стандартных. И эти ходы уже начинают появляться в партиях «живых» профессионалов. По мнению Фань Хуэя, этими ходами AlphaGo «освободила» го, показала, что многие продолжения, которые раньше считались плохими и никогда не игрались просто «из общих соображений», теперь станут вполне допустимыми.

9.5. Градиент по стратегиям и другие применения

> Виктор неторопливо поднялся с места, оглядел внимательно Электроника, словно выискивая в нем слабое место.
>
> — Превзойдет ли робот человека в обучении? — спросил он.
>
> — Это может случиться, — ответил Элек, — если человек сам перестанет учиться. Машине, между прочим, обучаться труднее, чем человеку, — добавил он.
>
> *Е. Велтистов. Новые приключения Электроника*

До сих пор практически все применения глубокого обучения с подкреплением, которые мы рассматривали, касались исключительно игр: одна из двух главных описанных выше работ о DQN училась играть в го, а другая — и вовсе в компьютерные игры из восьмидесятых. Надо сказать, что на победе в го применения обучения с подкреплением к играм вовсе не закончились; в частности, сейчас стоит ожидать ярких результатов в разных компьютерных играх. Любопытно, что хотя в таких играх, как *StarCraft* и *DotA 2*, не последнюю роль играют рефлексы, микроконтроль и умение точно рассчитать свои действия во времени и пространстве (когда закончится это заклинание, на какое расстояние стреляет морпех и т. п.), и в этом компьютер изначально имеет огромное преимущество над человеком, на самом деле пока в этих киберспортивных дисциплинах боты не могут составить никакой конкуренции людям. Так, на крупнейшем турнире по *DotA 2*, прошедшем в 2017 году *The International 7*, в качестве прорывного результата компания *DeepMind* представила бота, который смог победить лучших игроков в матче один на один — то есть в урезанной версии *DotA 2*, где собственно ничего кроме микроконтроля и нету. Но все-таки нам представляется, что успехи в киберспорте у компьютерных программ тоже не за горами: за *DotA 2* и *StarCraft* уже всерьез взялся *DeepMind*, так что дело обязательно пойдет.

Но есть ли более серьезные, «настоящие» применения для этих моделей? Конечно, есть! Первое и главное из них — *роботика*. Если попытаться обучить искусственную руку переносить вазу с одного стола на другой, неизбежно придется столкнуться с теми же проблемами, с которыми сталкивается обучающийся ходить ребенок из начала этой главы: мы не можем дать датасет с правильными поворотам шарниров искусственной руки, а можем только сказать, разбилась ли ваза. Кроме того, даже если тренировочные вазы не будут разбиваться, все равно очевидно, что миллионы попыток мы в реальной жизни сделать не сможем, то есть обучаться надо быстро. То же относится и, например, к вождению автомобилей: невозможно точно сказать, куда надо поворачивать руль в каждый конкретный момент, можно только сказать, что врезаться ни во что не надо. Поэтому неудивительно, что задачи роботики всегда были в центре обучения с подкреплением, а одна из первых и самых классических задач обучения с подкреплением — это задача балансировки шеста на платформе [33, 359].

Эти задачи отличаются от тех, которые мы до сих пор рассматривали, тем, что в них непрерывны не только состояния (в играх *Atari* состояний тоже было слишком много, чтобы их можно было подсчитать), но и *действия*: шарнир или руль можно повернуть на любой угол. Поэтому в этих задачах обычно используются не TD-обучение и Q-обучение, которые оказались так полезны для разнообразных игр, а другие методы.

В частности, центральный метод обучения с подкреплением в роботике — это *градиентный спуск по стратегиям* (policy gradient) [419]. Идея очень проста: рассмотрим стратегию π, которая делает действие a в зависимости от текущего состояния s и собственных параметров $\boldsymbol{\theta}$, $a \sim \pi(a \mid s; \boldsymbol{\theta})$. Мы хотим максимизировать некоторую целевую функцию J, например $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$; поскольку вознаграждения зависят от стратегии, а та задана параметрически, получается, что целевая функция — это функция от $\boldsymbol{\theta}$.

Давайте не будем строить никаких моделей окружающей среды, а просто будем напрямую оптимизировать целевую функцию по $\boldsymbol{\theta}$ градиентным спуском: подсчитаем $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ и сдвинемся в нужную сторону. На первый взгляд кажется, что сейчас мы не можем подсчитать градиент $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ аналитически, как мы делали раньше: слишком уж сложная это функция, ведь теперь в $J(\boldsymbol{\theta})$ спрятан долгий путь применения стратегии $\pi(\theta)$ по ходу целого эпизода, а то и вовсе уходящий на бесконечность. Один возможный выход — вычислять градиент приближенно, численно — давайте немножко поварьируем каждый из параметров θ_k и подсчитаем $\frac{\partial J}{\partial \theta_k} \approx \frac{J(\boldsymbol{\theta}+\epsilon \boldsymbol{u}_k)-J(\boldsymbol{\theta})}{\epsilon}$, где $\boldsymbol{u}_k$ — вектор из нулей с одной единицей в k-й компоненте. Это медленно (нужно вычислять $J(\boldsymbol{\theta} + \epsilon \boldsymbol{u}_k)$ отдельно для каждого параметра) и весьма неточно, но в некоторых приложениях работает неплохо.

Но есть, оказывается, способ лучше! Давайте все-таки посмотрим более пристально на функцию $J(\boldsymbol{\theta})$. Для простоты будем считать, что у нас марковский процесс принятия решений из одного шага, мы начинаем в состоянии $s \in S$ с вероятностью $p_0(s)$, и после одного действия $a \in A$ получаем награду R_s^a. Тогда:

$$J(\boldsymbol{\theta}) = \sum_{s \in S} p_0(s) \sum_{a \in A} \pi(a \mid s; \boldsymbol{\theta}) R_s^a, \text{ и}$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in S} p_0(s) \sum_{a \in A} R_s^a \nabla_{\boldsymbol{\theta}} \pi(a \mid s; \boldsymbol{\theta}).$$

Применим нехитрый трюк — заметим следующее равенство:

$$\nabla_{\boldsymbol{\theta}} \pi(a \mid s; \boldsymbol{\theta}) = \pi(a \mid s; \boldsymbol{\theta}) \frac{\nabla_{\boldsymbol{\theta}} \pi(a \mid s; \boldsymbol{\theta})}{\pi(a \mid s; \boldsymbol{\theta})} = \pi(a \mid s; \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}).$$

Это значит, что:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \sum_{s \in S} p_0(s) \sum_{a \in A} R_s^a \pi(a \mid s; \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}) = \mathbb{E}_{\pi(\boldsymbol{\theta})} \left[R_s^a \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}) \right].$$

Оказывается (этот факт так и называется — policy gradient theorem), что этот результат можно обобщить и на более продолжительные процессы. В целом:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi(\boldsymbol{\theta})} \left[\nabla_{\boldsymbol{\theta}} \log \pi(a \mid s; \boldsymbol{\theta}) Q^{\pi(\boldsymbol{\theta})}(s,a) \right].$$

Фактически это значит, что мы-таки можем подсчитать градиент функции $J(\boldsymbol{\theta})$. Так, например, классический алгоритм REINFORCE [571] просто берет результат $Q^{\pi(\boldsymbol{\theta})}(s,a)$ из текущего эпизода обучения (считая, что последний запуск агента представляет собой разумный сэмпл из этого распределения), а затем обновляет веса $\boldsymbol{\theta}$ при помощи градиента как показано выше. Это работает даже в случае бесконечного горизонта [35].

А более сложные методы строят модель окружающей среды (эта модель называется *критиком*), который служит для того, чтобы более точно оценивать $Q^{\pi(\boldsymbol{\theta})}(s,a)$ [427]. Глубокие нейронные сети здесь служат для того же самого, что и в играх: ими можно моделировать как стратегию $\pi(\boldsymbol{\theta})$, так и функцию $Q^{\pi(\boldsymbol{\theta})}(s,a)$. Чтобы сделать их обучение эффективным, правда, придется применить некоторые трюки, на которых мы сейчас не будем подробно останавливаться; в частности, нужно использовать разные варианты так называемого guided policy search [321], который лучше подходит для стратегий с очень большим числом параметров, как всегда и бывает в нейронных сетях.

Сейчас одно из важных направлений глубокого обучения с подкреплением в роботике — это непосредственный поиск стратегий, в котором нейронные сети обучаются целиком (end-to-end) и распознавать объекты в окружающем мире (компьютерным зрением, то есть по сути сверточными сетями), и представлять сами стратегии [143]. Для этого тоже можно адаптировать guided policy search [91, 413]. Современные подходы к глубокому обучению с подкреплением на основе градиентного спуска по стратегиям и других методов успешно применяется для того, чтобы искать выходы из лабиринтов [310], манипулировать объектами при помощи роботов с манипуляторами [110], двигаться в нужную сторону, руководствуясь визуальными данными (например, обучить дрон с видеокамерой следить за данным объектом) [316, 521], хватать объекты на основе данных с видеокамеры [120], и многое другое (см., например, подробный обзор [324]).

Есть и другие интересные применения. Например, в работе [201] DQN очень интересным образом применяется для того, чтобы улучшать порождение текста. В разделе 8.1 мы уже рассказывали об архитектурах типа encoder-decoder, в которых текст сначала кодируется в некоторое внутреннее представление, а затем декодируется обратно, возможно в виде ответа в диалоге или перевода на другой язык. А в [201] декодирование происходит итеративно:

- сначала декодер порождает первую версию выхода;
- в качестве списка возможных действий для DQN служит перечень слов, которые могут быть использованы, чтобы модифицировать текущее состояние выхода;

- и далее DQN пытается модифицировать выход так, чтобы в результате предложение стало лучше по отношению к целевой метрике качества.

Таким образом, DQN может самостоятельно выбирать, в каком порядке вносить изменения. На практике получается, что DQN сначала концентрируется на том, чтобы декодировать более простые и понятные части предложения, а потом использует их для того, чтобы уточнить более сложные места. В [201] эта идея применялась исключительно для того, чтобы построить автокодировщик текстов, то есть задача была в том, чтобы по поданному на вход предложению восстановить его же из сжатого представления кодировщика, а метрикой качества была BLEU-похожесть между входным и выходным предложением. Но аналогичный подход, скорее всего, можно применить и ко всем остальным задачам обработки естественного языка, где используются архитектуры encoder-decoder.

В целом, одним из важных трендов в современном обучении с подкреплением является то, что оно оказывается полезным не только для таких непосредственных приложений, как вождение автомобилей, манипуляция объектами или игра в го, но и для многих других задач, в том числе, казалось бы, обычных задач обучения с учителем. Нам кажется, что на этом пути нас ждет еще много интересных открытий, часть из которых, возможно, придется и на вашу долю, дорогие читатели. А нам пора двигаться дальше. В последней главе книги мы коснемся более теоретических материй... которые, впрочем, быстро окажутся очень даже прикладными.

Глава 10
Нейробайесовские методы,
или Прошлое и будущее машинного обучения

TL;DR

В этой достаточно математически насыщенной главе будет дано краткое и поверхностное введение в то, как сочетаются в современном машинном обучении нейронные сети и байесовский вывод. Мы:

- подробно разберем общий вид алгоритма EM для обучения моделей со скрытыми переменными;
- увидим на простых примерах основные идеи вариационных приближений;
- поговорим о конструкции вариационного автокодировщика, еще одной важной порождающей модели с глубокими нейронными сетями;
- отметим, как байесовский вывод помогает построить новые формы дропаута;
- закончим книгу нашими размышлениями о том, куда все это катится и что ждет нас в будущем.

10.1. Теорема Байеса и нейронные сети

> О святая математика, в общении с тобой хотел бы я провести остаток дней своих, забыв людскую злобу и несправедливость Вседержителя.
>
> *Лотреамон*

В главе 2 мы говорили о теореме Байеса и ее роли в машинном обучении, долго обсуждали пример с априорными и апостериорными распределениями броска монетки. А на протяжении всей книги не раз упоминали, что байесовский вывод в нейронных сетях тоже присутствует, и ссылались на какие-то загадочные вариационные приближения, которые якобы делают возможным вывод в очень сложных моделях. В этой главе мы на простых примерах поймем, что это такое. Но сначала давайте ненадолго вернемся к самой теореме Байеса:

$$p(\boldsymbol{\theta}|D) = \frac{p(\boldsymbol{\theta})p(D|\boldsymbol{\theta})}{p(D)} = \frac{p(\boldsymbol{\theta})p(D|\boldsymbol{\theta})}{\int_{\boldsymbol{\theta}\in\Theta} p(D \mid \boldsymbol{\theta})p(\boldsymbol{\theta})\mathrm{d}\boldsymbol{\theta}}.$$

В машинном обучении теорема Байеса позволяет из правдоподобия, которое обычно задается структурой модели, и априорного распределения, которое выражает наши представления об окружающем модель мире, получить апостериорное распределение, а затем, при желании, делать байесовские предсказания, интегрируя по всем возможным значениям параметров $\boldsymbol{\theta}$. С точки зрения нейронных сетей важно, что с помощью теоремы Байеса мы можем легко строить сложные вероятностные модели из простых, уточняя распределение скрытых параметров. Кроме того, выбирая априорное распределение, легко проводить регуляризацию. Более того, мы можем легко вводить модели с латентными (скрытыми) переменными и обучать их EM-алгоритмом или вариационными методами, как мы увидим в этой главе. Задача байесовской оценки непрерывных скрытых переменных — это что-то вроде задачи понижения размерности: надо хорошо восстановить латентные переменные, и они станут важными признаками, кратко описывающими каждый объект обучающей выборки.

Может показаться, что вся эта байесовская машинерия для нас не так уж важна, и можно обойтись максимальным правдоподобием. В главе 2 мы рассматривали пример подбрасывания монетки: конечно, если бросков монетки один-два, без байесовских методов получится ерунда, но после тысячи бросков уже не так важно, каким было априорное распределение. А когда мы говорим о глубоких нейронных сетях, наборы данных исчисляются даже не тысячами, а миллионами и миллиардами — сколько пикселей в ImageNet? Однако проблема в том, что роль априорного распределения определяется не общим числом точек N_{data}, а числом точек *на каждый свободный параметр* сети $N_{\text{data}}/N_{\text{model}}$. И в реальности оказывается, что как только N_{data} увеличивается, мы тут же хотим подтянуть к ней и N_{model}, чтобы

сделать модель выразительнее. Сверточные сети, которые обучаются на ImageNet, могут иметь десятки миллионов параметров. Так что на самом деле машинное обучение все время находится в «зоне актуальности» байесовских методов.

На этом месте достаточно подготовленный читатель может подумать, что дальше мы будем говорить о том, как добавить в нейронную сеть априорные распределения и как сделать байесовский вывод в нейронной сети. Это действительно заслуживающая внимания тема, и мы к ней вернемся в разделе 10.5. Байесовский подход на самом деле может дать возможность не только получить веса в локальном минимуме функции ошибки, но и оценить их дисперсию, а также и все апостериорное распределение $p(\boldsymbol{w} \mid D)$. Более того, в разделе 10.5 мы увидим, что такой подход дает совсем другой взгляд на дропаут, объяснив его с байесовской стороны.

Но по большей части далее мы будем говорить о совсем других вещах. Задача этой главы — постараться улучшить методы, использующиеся в байесовском выводе, особенно вариационные приближения к сложным апостериорным распределениям, с помощью «магии глубокого обучения». В качестве основных источников мы здесь рекомендуем исходную статью о байесовских автокодировщиках [280] и недавно появившееся подробное введение в тему [124].

Здесь появляется существенная разница в постановке задачи по сравнению со всем тем, что мы делали раньше. В машинном обучении модели часто имеют вероятностный смысл: мы обучаем не просто какую-то разделяющую поверхность, а распределение $p(\boldsymbol{x})$, которое показывает, насколько, по мнению обученной модели, вероятно появление той или иной точки в качестве точки данных. Уметь считать $p(\boldsymbol{x})$ вполне достаточно для того, чтобы решать, например, задачу классификации: чтобы распознавать рукописные цифры, мы обучаем десять моделей $p_0, p_1, \ldots, p_9$, а потом для классификации просто сравниваем $p_0(\boldsymbol{x}), p_1(\boldsymbol{x}), \ldots, p_9(\boldsymbol{x})$: какая вероятность больше, ту цифру мы и выдадим в качестве ответа.

А в разделе 8.2 мы говорили о том, что часто хочется не просто распознавать уже взятые откуда-то объекты, но и создавать их самостоятельно. Например, мы бы хотели научиться не только распознавать рукописные цифры на основе датасета MNIST, но и самостоятельно их генерировать, «писать от руки». Или генерировать картины в узнаваемом стиле известного художника, как в сервисах DeepArt и Prisma. Или... но здесь оставим читателю место, чтобы он мог дать волю собственной фантазии. Для этого нужно научиться не просто *вычислять* распределение $p(\boldsymbol{x})$, а *сэмплировать* из него, порождать точки по распределению $p(\boldsymbol{x})$.

Эта задача решалась, конечно, и в классическом байесовском обучении: есть, например, большая область методов Монте-Карло на основе марковских цепей (Markov chain Monte-Carlo), которые предназначены как раз для того, чтобы научиться сэмплировать из распределений, которые мы умеем только вычислять. Мы сейчас не будем углубляться в эту тему и отошлем читателя к соответствующим учебникам [44, 343, 381], а сами только скажем, что эти методы для очень сложных распределений (таких, как, например, распределение человеческих лиц в пространстве фотографий) работают крайне медленно или не работают вовсе.

В этой главе мы будем обучать модель сэмплировать (порождать точки) из сложного распределения с помощью нейронных сетей, которые будут здесь играть роль черного ящика для приближения какой угодно функции — в данном случае как раз плотности нужного распределения. Точнее говоря, нейронные сети будут обучаться преобразовывать какое-нибудь простое распределение, обычно нормальное со стандартными параметрами, в то, которое нам нужно. Если кратко описать предстоящую главу с математической точки зрения, мы рассмотрим способ строить приближения к сложным распределениям, в котором приближение берется из некоторого класса функций, а потом начнем в качестве этого класса функций подставлять нейронные сети. В результате получатся очень хорошо идейно мотивированные модели, которые многим исследователям представляются будущим глубокого обучения.

Сразу предупредим, что легко не будет — это, наверное, самая математически плотная глава во всей книге. С одной стороны, при первом чтении остаток главы, наверное, можно пропустить, да и на практике никто вас пока что не заставит применять именно эти методы. Но с другой стороны, эту главу можно считать своеобразной лакмусовой бумажкой: если вы можете ее прочесть и понять, у вас не должно возникнуть проблем с тем, чтобы читать самые современные статьи о глубоком обучении, и основную цель, которую мы ставим в этой книге, можно будет считать достигнутой. Итак, в путь!

10.2. Алгоритм EM

> Бедный Карлсон! Ему не позавидуешь. Алгоритм заставит его съесть пятьсот плюшек. Все до единой! И он наверняка умрет от обжорства.
>
> *В. Пароджанов.*
> *Дружелюбные алгоритмы, понятные каждому*

Чтобы применить байесовские методы к нейронным сетям (точнее, как мы увидим ниже, наоборот — нейронные сети к байесовским методам), сначала придется разобраться в том, как эти самые байесовские методы вообще работают. Мы начнем с того, что дадим байесовское обоснование одного из главных методов классического обучения без учителя, алгоритма Expectation-Maximization, который обычно называют просто EM-алгоритмом.

Идея EM-алгоритма довольно проста. Представьте себе, к примеру, задачу *кластеризации* на обычной плоскости $\mathbb{R}^2$: у вас есть набор точек, и вы хотите разбить их на подмножества так, чтобы внутри каждого из них точки были «похожи» друг на друга, то есть находились бы близко друг от друга, а точки из разных подмножеств как можно более существенно различались бы, то есть были бы далеки на плоскости.

Задачу кластеризации, конечно, можно решать очень многими разными способами [5, 573], и EM не всегда будет лучшим (как минимум он довольно медленный по сравнению с другими), но этот метод применим и к массе других задач, а кластеризация — хороший способ его проиллюстрировать.

Чтобы применить EM к кластеризации, нужно представить себе *все* данные о решении задачи кластеризации, которые могли бы у нас быть. В полном наборе данных у каждой точки будет написано, какие у нее координаты (это мы знали и раньше) и какому кластеру она должна принадлежать — а вот это как раз та информация, которую мы должны получить. В этом представлении один тестовый пример — это пара $(\boldsymbol{x}_i, z_i)$, где z_i показывает, какому кластеру принадлежит эта точка, и мы не знаем значений z_i.

Заметьте разницу: в «обычной» модели тоже всегда есть веса или параметры, которые мы пытаемся обучать, но их, как правило, относительно мало, меньше, чем данных, а тут мы видим, что на каждую точку в данных приходится целая своя переменная. Именно в таких ситуациях, когда в имеющихся данных некоторые переменные нам известны, а некоторые отсутствуют, и пригождается обычно EM-алгоритм.

Далее, чтобы применить EM-алгоритм, нужно сделать какие-то вероятностные предположения о том, как были порождены эти самые данные со скрытыми переменными. В случае кластеризации, проще говоря, нужно сделать предположения о форме кластеров. Давайте рассмотрим самый простой случай, когда точки лежат даже не на плоскости, а на прямой, и мы предполагаем, что они порождены двумя кластерами в виде нормальных распределений; более того, давайте предполагать, что дисперсия у этих распределений одинакова и равна σ: $x_1 \sim \mathcal{N}(x_1; \mu_1, \sigma^2)$ в первом кластере, $x_2 \sim \mathcal{N}(x_2; \mu_1, \sigma^2)$ во втором кластере.

А это значит, что распределение точек из обоих кластеров вместе представляет собой *смесь* двух нормальных распределений:

$$x \sim \alpha\mathcal{N}(x; \mu_1, \sigma^2) + (1 - \alpha)\mathcal{N}(x; \mu_2, \sigma^2),$$

где α показывает сравнительный вес кластеров друг относительно друга, то, насколько больше точек попадает в один кластер, чем в другой. То есть, математически говоря, наша задача состоит в том, чтобы найти параметры максимального правдоподобия (α, μ_1, μ_2) у этой смеси распределений.

Заметьте, что никаких скрытых переменных z пока что в модели нет. И мы сейчас можем записать эту задачу поиска параметров максимального правдоподобия в виде обычной задачи оптимизации:

$$\alpha, \mu_1, \mu_2 = \arg\max_{\alpha,\mu_1,\mu_2} \prod_{i=1}^{N} \Big(\alpha\mathcal{N}(x_i; \mu_1, \sigma^2) + (1 - \alpha)\mathcal{N}(x_i; \mu_2, \sigma^2)\Big),$$

где $x_1, \ldots, x_N$ — это точки из имеющегося набора данных.

Проблема здесь состоит в том, что эта функция слишком сложна; посмотрите — это огромное произведение сумм, и если раскрыть в нем скобки, слагаемых будет экспоненциально много. А если перейти к логарифмам, то мы получим большую сумму логарифмов сумм, что тоже не очень помогает. Как все это оптимизировать — непонятно.

Но если бы мы знали, какая точка из какого кластера была взята, C_1 или C_2, то легко подсчитали бы параметры максимального правдоподобия. Мы тогда просто разделили бы точки по соответствующим кластерам, и оценили среднее нормального распределения в каждом и долю каждого кластера:

$$\hat{\alpha} = \frac{|C_1|}{N}, \quad \hat{\mu}_1 = \frac{1}{|C_1|}\sum_{x_i \in C_1} x_i, \quad \hat{\mu}_2 = \frac{1}{|C_2|}\sum_{x_i \in C_2} x_i.$$

Или формально — давайте введем переменную z_i, которая равна нулю, если точка была взята из C_1, и единице, если из C_2. Тогда наш вероятностный процесс, порождающий точки, становится двухступенчатым, разбивается на две независимые части[1]: сначала мы случайно выбираем переменные z_i с вероятностью α, то есть общим правдоподобием:

$$p(Z \mid \alpha) = \prod_{i=1}^{N} \alpha^{z_i}(1-\alpha)^{1-z_i},$$

а потом с уже известными z_i набрасываем точки из соответствующих кластеров:

$$p(X \mid Z, \mu_1, \mu_2) = \prod_{i=1}^{N} \mathcal{N}(x; \mu_1, \sigma^2)^{z_i} \mathcal{N}(x; \mu_2, \sigma^2)^{1-z_i}.$$

Иначе говоря, мы теперь решаем такую задачу оптимизации (давайте для удобства сразу перейдем к логарифмам):

$$\alpha = \arg\max_{\alpha} \sum_{i=1}^{N} \left(z_i \ln \alpha + (1-z_i)\ln(1-\alpha)\right),$$

$$\mu_1, \mu_2 = \arg\max_{\mu_1,\mu_2} \sum_{i=1}^{N} \Big(z_i p(x; \mu_1, \sigma^2) + (1-z_i)\, p(x; \mu_2, \sigma^2)\Big),$$

где $p(x; \mu, \sigma)$ — плотность нормального распределения, $p(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(x-\mu)^2}$.

[1] Здесь можно было бы нарисовать графическую вероятностную модель, поговорить о d-разделимости, подробно обсудить, как условия d-разделимости связаны с условной независимостью в вероятностных моделях... но это все-таки увело бы нас очень далеко от предмета книги, поэтому мы просто оставим здесь несколько ссылок [44, 381, 594] и продолжим «на пальцах».

Обратите внимание, что теперь эта задача оптимизации распадается на три совершенно независимые и по отдельности довольно простые задачи:

- найти α как $\arg\max_\alpha \sum_{i=1}^N (z_i \ln\alpha + (1-z_i)\ln(1-\alpha))$; это всего лишь поиск параметра максимального правдоподобия для подбрасывания монетки, и найти оптимальное α легко: $\hat\alpha = \frac{|i:z_i=1|}{N} = \frac{|C_1|}{N}$;
- найти μ_1 как $\arg\max_{\mu_1} \sum_{i:z_i=1} p\left(x;\mu_1,\sigma^2\right)$ (обратите внимание, что в сумме только те слагаемые, для которых $z_i = 1$, зависят от μ_1, и — чудесное совпадение — они-то как раз и не зависят от μ_2); это всего лишь поиск параметра максимального правдоподобия для нормального распределения, и тут тоже μ_1 легко найти просто как среднее точек в первом кластере: $\hat\mu_1 = \frac{1}{|C_1|}\sum_{x_i\in C_1} x_i$;
- и, аналогично, зависящие от μ_2 слагаемые теперь не содержат μ_1, мы решаем задачу $\arg\max_{\mu_1} \sum_{i:z_i=0} p\left(x;\mu_2,\sigma^2\right)$ и находим $\hat\mu_2 = \frac{1}{|C_2|}\sum_{x_i\in C_2} x_i$.

На данный момент у нас получилось, что если z_i известны, то задача сразу разбивается на простые подзадачи и решается легко и непринужденно. Но что делать в реальной ситуации, когда z_i неизвестны?

Идея алгоритма EM состоит в том, чтобы *притвориться*, будто бы мы знаем z_i, а затем попеременно уточнять то скрытые переменные, то параметры модели. EM означает Expectation-Maximization, и в полном соответствии с названием алгоритм EM повторяет в цикле два шага:

- *E-шаг*: для известных параметров модели α,μ_1,μ_2 подсчитать ожидания скрытых переменных z_i; для кластеризации мы этого еще не делали, но это тоже несложно — если все параметры известны, то чтобы подсчитать, с какой вероятностью точка принадлежит тому или иному кластеру, нужно просто найти плотности распределений одного и другого кластера в этой точке:

$$\mathbb{E}\left[z_{ij}\right] = \frac{p(x=x_i|\mu=\mu_j)}{p\left(x=x_i \mid |\mu=\mu_1\right) + p\left(x=x_i \mid |\mu=\mu_2\right)} = \\ = \frac{e^{-\frac{1}{2\sigma^2}(x_i-\mu_j)^2}}{e^{-\frac{1}{2\sigma^2}(x_i-\mu_1)^2} + e^{-\frac{1}{2\sigma^2}(x_i-\mu_2)^2}};$$

- *M-шаг*: зафиксировать $z_i = \mathbb{E}\left[z_i\right]$ и пересчитать параметры модели по уже известным формулам:

$$\hat\alpha = \frac{|C_1|}{N}, \quad \hat\mu_1 = \frac{1}{|C_1|}\sum_{x_i\in C_1} x_i, \quad \hat\mu_2 = \frac{1}{|C_2|}\sum_{x_i\in C_2} x_i.$$

Получился итеративный алгоритм, который сначала инициализирует параметры модели случайными значениями (в случае кластеризации разумно выбрать две случайные точки из данных и объявить их центрами кластеров), а затем постепенно уточняет и параметры, и оценки скрытых переменных, пока не сойдется.

Итак, мы поняли, как работает EM-алгоритм, на примере задачи кластеризации. Мы получили некий метод, который выглядит разумно и дает хорошие результаты (по крайней мере на этой задаче), но звучит он пока что довольно подозрительно: с какой стати такое «вытягивание самого себя за уши» из случайных значений латентных переменных или параметров модели должно приводить к успеху? Давайте теперь разберемся не только в том, *как* работает EM в общем виде, но и *почему* он работает.

Для этого придется перейти к той самой обещанной «сложной математике». Сейчас мы дадим формальное обоснование алгоритма EM в общем виде; формул будет много, и для читателя, который раньше их не видел, у нас есть такой совет: постарайтесь «протащить» через эти формулы интуицию из примера, который мы не случайно так подробно рассмотрели. Пример с кластеризацией (то есть со смесью нормальных распределений) достаточно полно отражает все особенности задачи, и на нем можно понять и то, что происходит ниже.

Как и в примере с кластеризацией, мы будем решать задачу поиска параметров максимального правдоподобия $\boldsymbol{\theta}$ по данным $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$; правдоподобие можно записать так:

$$L(\boldsymbol{\theta} \mid \mathcal{X}) = p(\mathcal{X} \mid \boldsymbol{\theta}) = \prod p(\boldsymbol{x}_i \mid \boldsymbol{\theta}),$$

а максимизировать, понятное дело, можно не само правдоподобие, а его логарифм $\ell(\boldsymbol{\theta} \mid \mathcal{X}) = \log L(\boldsymbol{\theta} \mid \mathcal{X})$. EM может помочь, если этот максимум трудно найти аналитически.

Давайте предположим, что в данных есть *латентные переменные* (скрытые переменные, скрытые компоненты), причем модель устроена так, что если бы мы их знали, задача стала бы проще, а то и допускала бы аналитическое решение. На самом деле, совершенно не обязательно, чтобы эти переменные имели какой-то физический смысл, может быть, так просто удобнее, но физический смысл (такой, как номер кластера в предыдущем примере), конечно, обычно помогает их придумать. Например, представьте себе задачу порождения картинок с изображением цифр (мы рассмотрим этот пример в разделе 10.4). В этой задаче нам нужно породить всю картинку целиком так, чтобы она была «посвящена» одной и той же цифре, ведь половинка двойки плюс половинка восьмерки дадут скорее что-то похожее на рукописный твердый знак, чем на конкретную цифру. И это удобнее всего выразить именно так: сначала мы порождаем скрытую переменную z, которая соответствует тому, какую цифру хочется породить, а потом все распределение видимых данных $p(\boldsymbol{x} \mid z)$ будет уже зависеть от значения z [124].

В любом случае совместная плотность набора данных $\mathcal{Y} = (\mathcal{X}, \mathcal{Z})$ такова:

$$p(\boldsymbol{y} \mid \boldsymbol{\theta}) = p(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\theta}) = p(\boldsymbol{z} \mid x, \boldsymbol{\theta}) p(x \mid \boldsymbol{\theta}),$$

и полное правдоподобие данных теперь составляет $L(\boldsymbol{\theta} \mid \mathcal{Z}) = p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta})$.

Это случайная величина (потому что $\mathcal{Y}$ неизвестно), а настоящее правдоподобие можно вычислить как ожидание полного правдоподобия по скрытым переменным $\mathcal{Y}$:

$$L(\boldsymbol{\theta}) = \mathbb{E}_{\mathcal{Y}}\left[p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta}) \mid \mathcal{X}, \boldsymbol{\theta}\right].$$

Теперь E-шаг алгоритма EM вычисляет условное ожидание (логарифма) полного правдоподобия при условии $\mathcal{X}$ и текущих оценок параметров $\boldsymbol{\theta}_n$:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n) = \mathbb{E}\left[\log p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta}) \mid \mathcal{X}, \boldsymbol{\theta}_n\right].$$

Здесь $\boldsymbol{\theta}_n$ — текущие оценки параметров модели, а $\boldsymbol{\theta}$ — неизвестные значения, которые мы хотим получить в конечном счете; это значит, что $Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n)$ — это функция от $\boldsymbol{\theta}$. Условное ожидание можно переписать так:

$$E\left[\log p(\mathcal{X}, \mathcal{Y} \mid \boldsymbol{\theta}) \mid \mathcal{X}, \boldsymbol{\theta}_n\right] = \int_{\boldsymbol{z}} \log p(\mathcal{X}, \boldsymbol{z} \mid \boldsymbol{\theta}) p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \mathrm{d}\boldsymbol{z},$$

где $p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)$ — маргинальное распределение скрытых компонентов данных.

EM лучше всего применять, когда это выражение легко подсчитать, может быть, даже можно подсчитать аналитически. Вместо $p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)$ можно подставить $p(\boldsymbol{z}, \mathcal{X} \mid \boldsymbol{\theta}_n) = p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) p(\mathcal{X} \mid \boldsymbol{\theta}_n)$, от этого ничего не изменится.

В итоге после E-шага алгоритма EM мы получаем функцию $Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n)$. А на M-шаге максимизируем функцию Q по параметрам модели, получая новую оценку $\boldsymbol{\theta}_{n+1}$:

$$\boldsymbol{\theta}_{n+1} = \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n).$$

Затем эта процедура повторяется до сходимости.

Осталось понять, что же, собственно, означает эта таинственная функция $Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n)$ и почему все это работает.

Мы хотели перейти от $\boldsymbol{\theta}_n$ к $\boldsymbol{\theta}$, для которого $\ell(\boldsymbol{\theta}) > \ell(\boldsymbol{\theta}_n)$. Давайте оценим разность этих двух величин. Дальше мы будем вести достаточно длинную цепочку равенств, которые будем по ходу комментировать:

$$\ell(\boldsymbol{\theta}) - \ell(\boldsymbol{\theta}_n) = \ldots$$

(по определению и одной из предыдущих формул)

$$\ldots = \log\left(\int_{\boldsymbol{z}} p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta}) \mathrm{d}\boldsymbol{z}\right) - \log p(\mathcal{X} \mid \boldsymbol{\theta}_n) = \ldots$$

(домножим и разделим на $p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)$)

$$\ldots = \log\left(\int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \frac{p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)} \mathrm{d}\boldsymbol{z}\right) - \log p(\mathcal{X} \mid \boldsymbol{\theta}_n) \geq \ldots$$

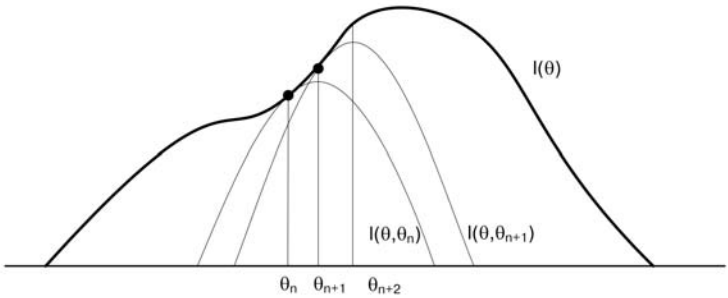


Рис. 10.1. Алгоритмы миноризации-максимизации

(по неравенству Йенсена: логарифм — выпуклая функция, а математическое ожидание — линейное преобразование, так что $\log \mathbb{E}_{p(x)} f(x) \geq \mathbb{E}_{p(x)} \log f(x)$, а у нас берется ожидание по распределению $p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)$)

$$\ldots \geq \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z},\boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z} - \log p(\mathcal{X} \mid \boldsymbol{\theta}_n) = \ldots$$

(занесем теперь $p(\mathcal{X} \mid \boldsymbol{\theta}_n)$ под интеграл и под логарифм — в нем $\boldsymbol{z}$ не участвует, так что для математического ожидания по $\boldsymbol{z}$ это просто константа)

$$= \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z},\boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\mathcal{X} \mid \boldsymbol{\theta}_n) p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z}.$$

Итак, мы получили, что

$$\ell(\boldsymbol{\theta}) \geq l(\boldsymbol{\theta},\boldsymbol{\theta}_n) = \ell(\boldsymbol{\theta}_n) + \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z},\boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\mathcal{X} \mid \boldsymbol{\theta}_n) p(\boldsymbol{z} \mid \mathcal{X},\boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z}.$$

Кроме того, легко доказать, что $l(\boldsymbol{\theta}_n,\boldsymbol{\theta}_n) = \ell(\boldsymbol{\theta}_n)$. Иначе говоря, мы нашли нижнюю оценку для функции $\ell(\boldsymbol{\theta})$, которая касается, оказывается равной $\ell(\boldsymbol{\theta})$ в точке $\boldsymbol{\theta}_n$. Это значит, что в рамках EM-алгоритма мы сначала нашли нижнюю оценку для функции правдоподобия, которая касается самого правдоподобия в текущей точке, а потом сместились туда, где она максимальна; естественно, при этом сама функция правдоподобия тоже увеличится, ей просто деваться некуда (рис. 10.1). Такая общая схема оптимизации, при которой максимизируют нижнюю оценку на оптимизируемую функцию, иногда еще называется *MM-алгоритмом*, от слов Minorization-Maximization.

Осталось только понять, что максимизировать можно Q. И снова придется вытерпеть цепочку равенств:

$$\boldsymbol{\theta}_{n+1} = \arg\max_{\boldsymbol{\theta}} l(\boldsymbol{\theta},\boldsymbol{\theta}_n) = \ldots$$

(перепишем по формуле выше)

$$= \ldots \arg\max_{\boldsymbol{\theta}} \left\{ \ell(\boldsymbol{\theta}_n) + \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \log \left(\frac{p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\mathcal{X} \mid \boldsymbol{\theta}_n) p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n)} \right) \mathrm{d}\boldsymbol{z} \right\} = \ldots$$

(выбросим из-под arg max все, что не зависит от $\boldsymbol{\theta}$ — от этого точка, где достигается максимум, не изменится)

$$= \ldots \arg\max_{\boldsymbol{\theta}} \left\{ \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \log \left(p(\mathcal{X} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta}) \right) \mathrm{d}\boldsymbol{z} \right\} = \ldots$$

(свернем опять произведение $p(x \mid \boldsymbol{z})p(\boldsymbol{z})$ в совместную вероятность $p(x, \boldsymbol{z})$)

$$= \ldots \arg\max_{\boldsymbol{\theta}} \left\{ \int_{\boldsymbol{z}} p(\boldsymbol{z} \mid \mathcal{X}, \boldsymbol{\theta}_n) \log p(\mathcal{X}, \boldsymbol{z} \mid \boldsymbol{\theta}) \mathrm{d}\boldsymbol{z} \right\} = \arg\max_{\boldsymbol{\theta}} \left\{ Q(\boldsymbol{\theta}, \boldsymbol{\theta}_n) \right\}.$$

Вот и получился EM-алгоритм.

Заметим, что для того, чтобы EM-алгоритм сработал, в принципе достаточно просто находить $\boldsymbol{\theta}_{n+1}$, для которого $Q(\boldsymbol{\theta}_{n+1}, \boldsymbol{\theta}_n) > Q(\boldsymbol{\theta}_n, \boldsymbol{\theta}_n)$: чтобы $\ell(\boldsymbol{\theta})$ увеличивалось, не обязательно находить именно максимум ее нижней оценки, достаточно сместиться туда, где нижняя оценка просто хоть на сколько-нибудь увеличится. Такая схема называется *обобщенным EM-алгоритмом* (Generalized EM).

10.3. Вариационные приближения

> Бах решил, что тут, пожалуй, лучше всего подойдут вариации, хотя до сих пор считал, что это дело неблагодарное... Тем не менее, эти вариации, как и все, что он создавал в то время, получились у него великолепными... Граф называл этот цикл своими вариациями. Он никак не мог ими насладиться, и долго еще, как только у него начиналась бессонница, он, бывало, говорил: «Любезный Гольдберг, сыграй-ка мне какую-нибудь из моих вариаций».
>
> *И. Форкель. О жизни, искусстве и о произведениях Иоганна Себастьяна Баха*

Следующая важная идея байесовского вывода, которую нам нужно осознать и для которой, собственно, и предназначены описанные в этой главе методы, — это *вариационные приближения*. Их рассмотрение будет очень похоже на то, что мы говорили о EM-алгоритме, и мы сможем еще раз обсудить всю эту ситуацию со скрытыми параметрами с немножко другого угла зрения.

Мы по-прежнему находимся в той же общей ситуации, что и в предыдущем разделе, где нам помогал EM-алгоритм: предположим, что набор данных $X = \{\boldsymbol{x}_i\}_{i=1}^N$ был порожден двухступенчатым процессом с параметрами модели $\boldsymbol{\theta}$: сначала из

какого-то априорного распределения $p(\boldsymbol{z} \mid \boldsymbol{\theta})$ породили вектор скрытых переменных $\boldsymbol{z}$, а затем из условного распределения $p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})$ породили собственно точку $\boldsymbol{x}$. Мы знаем распределения $p(\boldsymbol{z} \mid \boldsymbol{\theta})$ и $p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})$, но распределение $p(\boldsymbol{x} \mid \boldsymbol{\theta})$:

$$p(\boldsymbol{x} \mid \boldsymbol{\theta}) = \int_Z p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta}) \mathrm{d}\boldsymbol{z}$$

для нас слишком сложное, мы не можем его подсчитать напрямую.

EM-алгоритм применялся в ситуациях, когда мы можем либо аналитически, либо численно подсчитать распределение скрытых переменных при фиксированных параметрах $p(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta})$; обычно его можно подсчитать по теореме Байеса:

$$p(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}) = \frac{p(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\theta}) p(\boldsymbol{z} \mid \boldsymbol{\theta})}{p(\boldsymbol{x} \mid \boldsymbol{\theta})}.$$

Тогда мы считаем это распределение на E-шаге EM-алгоритма, получаем оценки ожиданий $\boldsymbol{z}$, максимизируем целевую функцию на M-шаге по $\boldsymbol{\theta}$ с фиксированными оценками и так далее, как в предыдущем разделе.

Но что если даже распределение $p(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta})$ подсчитать не получается? Это не такой уж редкий случай: представим себе, например, что распределение $p(\boldsymbol{x} \mid \boldsymbol{z},\boldsymbol{\theta})$ задано привычной нам нейронной сетью. Даже обычная двухслойная нейронная сеть с нелинейным скрытым уровнем — это настолько сложное распределение, что просто взять и умножить его на априорное распределение $p(\boldsymbol{z} \mid \boldsymbol{\theta})$ не получится.

В математическом моделировании есть универсальный ответ на все подобные затруднения: если структура объекта слишком сложна, значит, нужно просто приблизить его с помощью какого-нибудь более простого объекта. Суть вариационного метода при этом состоит в том, что более простое приближение мы ищем среди некоторого класса функций (распределений), и пытаемся найти в этом более простом классе элемент, который наиболее похож на тот «сложный» элемент, который мы приближаем. Если этот класс будет задан параметрически, значит, у нас появятся новые *вариационные параметры*, по которым мы, будем надеяться, сможем оптимизировать приближение. Впрочем, одна из ключевых особенностей метода, который мы сейчас рассматриваем, как раз в том, что приближение мы ищем вовсе не параметрически.

Однако в целом смысл происходящего точно такой же: мы приближаем сложное распределение более простой формой, выбирая то распределение из более простого семейства, которое лучше всего приближает сложное распределение. «Похожесть» можно определить по расстоянию Кульбака — Лейблера:

$$\mathrm{KL}(p\|q) = \int p(x) \ln \frac{p(x)}{q(x)} \mathrm{d}x = -\int p(x) \ln \frac{q(x)}{p(x)} \mathrm{d}x.$$

Чтобы объяснить, как все это работает в реальности, вернемся сначала к одному из взглядов на EM-алгоритм: пусть X — наблюдаемые переменные, а Z —

латентные. EM-алгоритм предполагает, что мы умеем считать плотность совместного распределения $p(X, Z \mid \boldsymbol{\theta})$, а хотим максимизировать $p(X \mid \boldsymbol{\theta})$. Они связаны, например, так:

$$p(X, Z \mid \boldsymbol{\theta}) = p(X \mid \boldsymbol{\theta})p(Z \mid X,\boldsymbol{\theta}),$$

а значит,

$$\ln p(X \mid \boldsymbol{\theta}) = \ln p(X, Z \mid \boldsymbol{\theta}) - \ln p(Z \mid X,\boldsymbol{\theta}).$$

Рассмотрим новое распределение $q(Z)$, которое и будет служить тем самым приближением. Давайте просто механически добавим его в наши формулы. Рассмотрим формулу выше и возьмем в ее левой и правой частях ожидание по распределению $q(Z)$:

$$\int_Z q(Z) \ln p(X \mid \boldsymbol{\theta})\mathrm{d}Z = \int_Z q(Z) \ln p(X, Z \mid \boldsymbol{\theta})\mathrm{d}Z - \int_Z q(Z) \ln p(Z \mid X,\boldsymbol{\theta})\mathrm{d}Z.$$

Но $\ln p(X \mid \boldsymbol{\theta})$ от Z не зависит, так что ожидание слева равно самому $\ln p(X \mid \boldsymbol{\theta})$:

$$\ln p(X \mid \boldsymbol{\theta}) = \int_Z q(Z) \ln p(X, Z \mid \boldsymbol{\theta})dZ - \int_Z q(Z) \ln p(Z \mid X, \boldsymbol{\theta})dZ.$$

А теперь справа давайте под интегралом добавим и вычтем $q(Z) \ln q(Z)$ (да, это выглядит как непонятная магия — спокойствие, сейчас все прояснится); в результате у нас получится:

$$\begin{aligned}\ln p(X \mid \boldsymbol{\theta}) &= \int_Z q(Z) \left(\ln p(X, Z \mid \boldsymbol{\theta}) - \ln q(Z) + \ln q(Z) - \ln p(Z \mid X, \boldsymbol{\theta})\right) dZ \\ &= \int_Z q(Z) \ln \frac{p(X, Z \mid \boldsymbol{\theta})}{q(Z)} dZ - \int_Z q(Z) \ln \frac{p(Z \mid X, \boldsymbol{\theta})}{q(Z)} dZ = \\ &= \mathcal{L}(q, \boldsymbol{\theta}) + \mathrm{KL}(q \| p(Z \mid X, \boldsymbol{\theta})),\end{aligned}$$

где $\mathcal{L}(q, \boldsymbol{\theta}) = \int_Z q(Z) \ln \frac{p(X,Z|\boldsymbol{\theta})}{q(Z)} dZ$ — некий функционал уже от $p(X,Z \mid \boldsymbol{\theta})$, от совместного распределения, которое, по нашему предположению, достаточно простое.

Посмотрите, что у нас получилось: $\mathrm{KL}(q \| p(Z \mid X, \boldsymbol{\theta}))$ — это расстояние Кульбака — Лейблера; оно всегда неотрицательно, и равно нулю только если $q = p$. Поэтому $\mathcal{L}(q, \boldsymbol{\theta})$ — это нижняя оценка на $\ln p(X \mid \boldsymbol{\theta})$. И в EM-алгоритме, получается, мы для текущего $\boldsymbol{\theta}_n$:

- на E-шаге максимизируем $\mathcal{L}(q, \boldsymbol{\theta}_n)$ по q для фиксированного $\boldsymbol{\theta}_n$; максимум достигается, когда $\mathrm{KL}(q \| p(Z \mid X, \boldsymbol{\theta})) = 0$, то есть когда $q(Z) = p(Z \mid X, \boldsymbol{\theta}_n)$;
- на M-шаге фиксируем $q(Z)$ и максимизируем нижнюю оценку правдоподобия $\mathcal{L}(q, \boldsymbol{\theta})$ по $\boldsymbol{\theta}$, получая $\boldsymbol{\theta}_{n+1}$.

Нижняя оценка $\mathcal{L}(q, \boldsymbol{\theta})$ — это один из важнейших объектов в теории вариационных приближений. Она называется *вариационной нижней оценкой* (variational

lower bound, evidence lower bound, ELBO). А сами вариационные методы работают так: пусть мы хотим максимизировать $p(X)$ со скрытыми переменными Z, и $p(Z \mid X)$ — сложное распределение. Давайте искать приближение к нему в виде распределения более простой структуры $q(Z)$. Тогда по тем же соображениям

$$\ln p(X) = \mathcal{L}(q) + \mathrm{KL}(q \| p(Z \mid X)), \text{ где}$$

$$\mathcal{L}(q) = \int_Z q(Z) \ln \frac{p(X,Z)}{q(Z)} dZ, \quad \mathrm{KL}(q\|p) = -\int_Z q(Z) \ln \frac{p(Z \mid X)}{q(Z)} dZ.$$

И мы снова можем максимизировать вариационную нижнюю оценку $\mathcal{L}(q)$, приближая $q(Z)$ к $p(Z \mid X)$. Здесь получился редкий частный случай функциональной оптимизации: казалось бы, нам нужно было решить нереально сложную задачу, максимизировать $\mathcal{L}(q, \boldsymbol{\theta}_n)$ по функции, по распределению q. Но в результате нехитрых преобразований оказалось, что для этого достаточно всего лишь подсчитать $q(Z) = p(Z \mid X, \boldsymbol{\theta}_n)$, или хотя бы приблизить $q(Z) \approx p(Z \mid X, \boldsymbol{\theta}_n)$. Если бы мы брали $q(Z)$ из параметрического семейства, то поиск $q(Z)$ свелся бы к (возможно, приближенному) решению некоторой задачи оптимизации, но прелесть вариационных приближений в том, что часто параметрическое семейство и вовсе не нужно!

Рассмотрим сначала главный частный случай вариационных приближений — случай, когда q полностью факторизуется, раскладывается на множители, зависящие от одной переменной или непересекающихся подмножеств переменных:

$$q(Z) = \prod_{i=1}^{M} q_i(Z_i), \quad Z_i \cap Z_j = \emptyset \text{ для всех } i,j.$$

Ключевой момент здесь в том, что мы никак не ограничиваем q_i! Никаких предположений о форме распределений, никакого параметрического семейства. Однако теперь мы просто должны максимизировать $\mathcal{L}(q)$, сделав предположение о форме $q(Z)$: $q(Z) = \prod_{i=1}^{M} q_i(Z_i)$. Оставим то, что зависит от одного фактора q_j:

$$\begin{aligned}
\mathcal{L}(q) &= \int q \ln \frac{p(X,Z)}{q(Z)} dZ = \int \prod_{i=1}^{M} q_i \left(\ln p(X,Z) - \sum_i \ln q_i \right) dZ = \\
&= \int q_j \left[\int \ln p(X,Z) \prod_{i \neq j} q_i dZ_i \right] dZ_j - \int q_j \ln q_j dZ_j + \text{const} = \\
&= \int q_j \ln \tilde{p}(X, Z_j) dZ_j - \int q_j \ln q_j dZ_j + \text{const},
\end{aligned}$$

где $\ln \tilde{p}(X, Z_j) = \mathbb{E}_{i \neq j} [\ln p(X,Z)] + \text{const}$.

Как теперь максимизировать такую величину:

$$\mathcal{L}(q) = \int q_j \ln \tilde{p}(X, Z_j) dZ_j - \int q_j \ln q_j dZ_j + \text{const}$$

по q_j при фиксированных q_i, $i \neq j$? Эта формула – просто KL-расстояние между $q_j(Z_j)$ и $\tilde{p}(X, Z_j)$! Значит, можно брать q^* так, чтобы:

$$\ln q^*(Z_j) = \mathbb{E}_{i \neq j} \left[\ln p(X, Z)\right] + \text{const},$$

а эта задача обычно решается достаточно просто.

Давайте разберем конкретный пример: приблизим двумерное нормальное распределение произведением одномерных. Рассмотрим двумерный гауссиан:

$$p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z} \mid \boldsymbol{\mu}, \Lambda^{-1}), \quad \boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \quad \Lambda = \begin{pmatrix} \lambda_{11} & \lambda_{12} \\ \lambda_{12} & \lambda_{22} \end{pmatrix},$$

$$p(\boldsymbol{z}) = \frac{1}{2\pi|\Lambda|} e^{-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{\mu})^\top \Lambda (\boldsymbol{z}-\boldsymbol{\mu})}.$$

Это распределение не раскладывается на независимые множители, но давайте приблизим его распределением, которое раскладывается: $q(\boldsymbol{z}) = q_1(z_1)q_2(z_2)$.

Воспользуемся выведенной выше формулой $\ln q^*(Z_j) = \mathbb{E}_{i \neq j} \left[\ln p(X, Z)\right] + \text{const}$; когда мы подсчитываем $q_1(z_1)$, в const попадает все, что не зависит от z_1:

$$\begin{aligned} \ln q_1^*(z_1) &= \mathbb{E}\left[\ln p(\boldsymbol{z})\right] + \text{const} = \mathbb{E}\left[-\frac{1}{2}(z-\mu)^\top \Lambda (z-\mu)\right] + \text{const} = \\ &= \mathbb{E}\left[-\frac{1}{2}(z_1-\mu_1)^\top \lambda_{11}(z_1-\mu_1) - (z_1-\mu_1)^\top \lambda_{12}(z_2-\mu_2)\right] + \text{const} = \\ &= -\frac{1}{2}\lambda_{11} z_1^2 + \left(\mu_1\lambda_{11} - \left(\mathbb{E}\left[z_2\right] - \mu_2\right)\right) z_1 + \text{const}. \end{aligned}$$

Смотрите — в качестве приближения у нас получилось нормальное распределение, причем получилось само по себе, без нашего участия! Выделяя полный квадрат:

$$q_1^*(z_1) = \mathcal{N}(z_1 \mid m_1, \lambda_{11}^{-1}), \quad \text{где} \quad m_1 = \mu_1 - \lambda_{11}^{-1}\lambda_{12}(\mathbb{E}\left[z_2\right] - \mu_2).$$

Аналогично,

$$q_2^*(z_2) = \mathcal{N}(z_2 \mid m_2, \lambda_{22}^{-1}), \quad \text{где} \quad m_2 = \mu_2 - \lambda_{22}^{-1}\lambda_{12}(\mathbb{E}\left[z_1\right] - \mu_1).$$

Здесь $\mathbb{E}\left[z_1\right] = m_1$, $\mathbb{E}\left[z_2\right] = m_2$, и нам надо просто решить систему; получится, естественно, $m_1 = \mu_1$, $m_2 = \mu_2$.

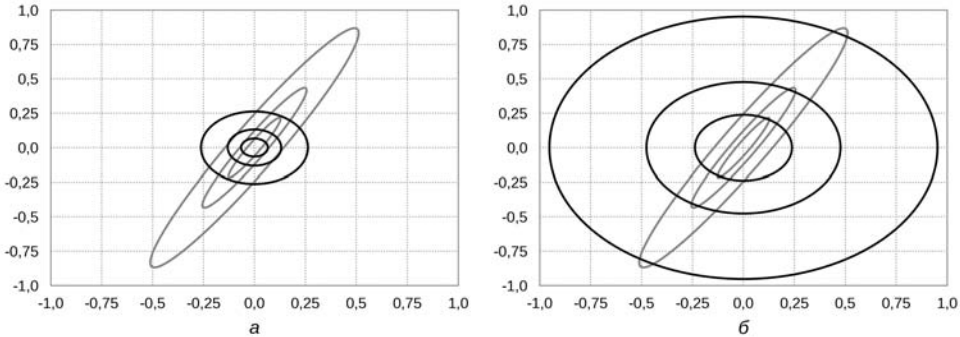


Рис. 10.2. Вариационные приближения к двумерному гауссиану

Сравним теперь это с обычным маргиналом (проекцией): на рис. 10.2, *а* мы изобразили полученное вариационное приближение, а на рис. 10.2, *б* — произведение проекций на оси X и Y. Мы видим здесь тот самый эффект, который мы обсуждали в разделе 8.5 (вспомните рис. 8.11) для разных вариантов расстояния Кульбака — Лейблера: в то время как произведение маргиналов дает широкое «накрывающее» распределение, вариационные приближения выбирают один конкретный пик, что для задач машинного обучения обычно предпочтительнее.

А важный для нашего дальнейшего рассмотрения конкретный пример — это *метод главных компонент* (principal components analysis, PCA). Это простейшая модель с непрерывными латентными переменными; возможно, вы слышали о ней в «обычных» курсах статистики или машинного обучения. Смысл PCA состоит в том, чтобы сократить размерность, потеряв как можно меньше информации о датасете; формально это значит, что мы пытаемся найти такое подпространство меньшей размерности, проекция на которую сохраняет как можно большую часть дисперсии исходного набора точек. Алгоритмически обычный PCA сводится к диагонализации эмпирической ковариационной матрицы датасета, то есть к подсчету ее собственных векторов.

Но сейчас нас интересует байесовский взгляд на то, что происходит в методе главных компонент. Итак, пусть наши исходные объекты находятся в пространстве размерности D, а латентные переменные — в пространстве размерности d: $X \in \mathbb{R}^D$, $Z \in \mathbb{R}^d$. Запишем распределения:

$$p(X,Z \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i,z_i \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i \mid z_i,\boldsymbol{\theta})p(z_i \mid \boldsymbol{\theta}).$$

Будем предполагать, что латентные переменные z_i берутся из нормального распределения вокруг нуля, а собственно наблюдаемые переменные получаются из них линейным преобразованием с нормально распределенным шумом:

$$p(X,Z \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} \mathcal{N}(x_i \mid \mu + Wz_i, \sigma I)\mathcal{N}(z_i \mid 0, I).$$

Здесь матрица W размерности $D \times d$ и вектор $\mu \in \mathbb{R}^d$ — это и есть искомые веса. Задача та же: максимизировать $p(X \mid \boldsymbol{\theta})$ по $\boldsymbol{\theta}$. Ее можно, конечно, решить явно, как в исходном методе главных компонент, а можно и EM-алгоритмом: на E-шаге искать $p(z_i \mid x_i, \boldsymbol{\theta})$, а на M-шаге максимизировать $\mathbb{E}_z \log p(X,Z \mid \boldsymbol{\theta})$; в этом простом частном случае и E-, и M-шаг можно выразить явно, аналитически.

Казалось бы, странно: зачем запускать итеративный процесс, если можно просто явно все посчитать через собственные векторы? Оказывается, что иногда так действительно лучше: сложность метода главных компонент в явном виде составляет $O(nD^2 + D^3)$ (искать собственные векторы у больших матриц недешево!), а сложность одной итерации EM-алгоритма — всего лишь $O(nDd)$. При маленьких d это гораздо лучше, и итераций можно себе позволить довольно много.

Есть и другие преимущества. Например, что делать, если во входных данных, в X, попадаются пропущенные данные? PCA нужна правильная плотная матрица, строки или столбцы с неизвестными данными придется выкидывать, а в таком случае мы рискуем остаться без данных совершенно.

Или предположим, что часть скрытых переменных нам уже известна (обучение с частичным привлечением учителя): опять же, в явном алгоритме непонятно, что делать, а тут совершенно понятно, надо просто зафиксировать известные z_i и не пересчитывать их потом.

Еще более интересное байесовское обобщение идеи PCA — это смесь нескольких подпространств главных компонент (mixture of PCA). Идея такая: что если данные не укладываются вдоль одного линейного подпространства малой размерности, но укладываются вдоль нескольких? В базовом PCA совершенно непонятно, как ввести такое расширение. А при вероятностном подходе все получается само собой: мы добавляем скрытые переменные t_i, которые показывают, из какого именно линейного подпространства (из нескольких) выбирается точка, и общее совместное распределение теперь выглядит так:

$$p(X,Z,T \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i \mid t_i, z_i, \boldsymbol{\theta}) p(z_i \mid \boldsymbol{\theta}) p(t_i \mid \boldsymbol{\theta}).$$

Распределения $p(x_i \mid t_i, z_i, \boldsymbol{\theta})$ выглядят точно так же, как $p(x_i \mid z_i, \boldsymbol{\theta})$ выше, но параметры выбираются соответственно значению t_i. Теперь можно точно так же вести вывод EM-алгоритмом, но метод уже получается не совсем линейный!

Однако даже такие вариации метода главных компонент в любом случае предполагают линейность базового подпространства. Но в жизни поверхности часто нам достаются совершенно нелинейные: согласитесь, вряд ли «настоящие фотографии» образуют линейное подпространство в пространстве всех изображений. Для того, чтобы их выразить, нам и помогут нейронные сети...

10.4. Вариационный автокодировщик

> Здесь следует обратить внимание на то, что прежде всего я могу мыслить самого себя, это аподиктическое Ego, совсем иначе, в свободной вариации, и таким образом получить систему возможных вариаций себя самого, причем любая такая вариация уничтожается любой другой и тем Ego, которым я действительно являюсь. Это есть система априорной несовместимости.
>
> *Э. Гуссерль. Картезианские размышления*

Вариационный автокодировщик [124, 280] — это недавно появившийся вариант порождающих моделей (вспомните раздел 8.2), который основан на вариационных приближениях. И на самом деле основная цель процесса здесь будет ровно та же, что и в соперничающих автокодировщиках (AAE; см. раздел 8.5): мы хотим сделать так, чтобы распределение скрытых факторов в автокодировщике было похоже на какое-нибудь стандартное распределение (например, нормальное), что затем поможет нам сэмплировать подходящие скрытые факторы и разворачивать их в правдоподобные объекты декодирующей частью.

Однако теперь мы будем идти к этой цели совсем другим путем. Содержательно идея вариационного автокодировщика поразительно похожа на то, что мы делали выше для метода главных компонент. Давайте просто вместо линейной функции подставим нелинейную:

$$p(X,Z \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i,z_i \mid \boldsymbol{\theta}) = \prod_{i=1}^{m} p(x_i \mid z_i,\boldsymbol{\theta})p(z_i \mid \boldsymbol{\theta}) = \prod_{i=1}^{m}\prod_{j=1}^{D} \mathcal{N}(x_{ij} \mid \mu_j(z_i),\sigma_j(z_i))\mathcal{N}(z_i \mid 0, I).$$

Обратите внимание: это ровно то же самое, что в PCA. Точно так же мы предполагаем, что латентные переменные берутся из стандартного нормального распределения, но теперь функции $\mu_j(z_i)$ и $\sigma_j(z_i)$ могут быть какими угодно. Как можно задать «какие угодно» функции? Конечно же, нейронной сетью! У нас появляется нейронная сеть, которая берет на вход z и производит те самые μ и σ, а параметры $\boldsymbol{\theta}$ теперь становятся просто параметрами этой нейронной сети.

Конечно, явным образом задачу максимизации теперь решить не получится. Но и с EM-алгоритмом не все так просто: на E-шаге нужно рассчитать $p(z_i \mid x_i,\boldsymbol{\theta})$, и это аналитически сделать не получится:

$$p(z_i \mid x_i,\boldsymbol{\theta}) = \frac{p(x_i \mid z_i,\boldsymbol{\theta})p(z_i)}{\int p(x_i \mid z_i,\boldsymbol{\theta})p(z_i)},$$

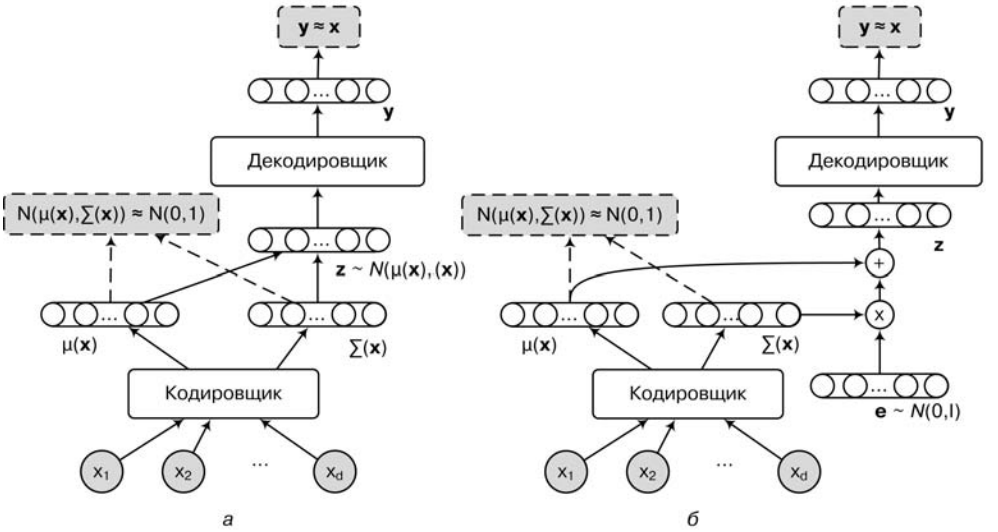


Рис. 10.3. Структура вариационного автокодировщика: *а* — основная идея; *б* — реализация с помощью репараметризации

и интеграл в знаменателе не берется. Что делать? Нужно вспомнить, что EM-алгоритм максимизирует нижнюю оценку на $q(Z)$. Теперь не получается оптимизировать функционально, по всем-всем-всем $q(Z)$, ну так давайте теперь просто параметризуем $q(Z)$ и будем решать задачу параметрической оптимизации.

С этим получается так: хочется взять $q(Z)$ побогаче (чтобы лучше приближать), но при этом надо все равно уметь оптимизировать. Чтобы этого добиться, давайте в качестве $q(Z)$... тоже возьмем нейронную сеть! С параметрами ϕ:

$$q(Z \mid X,\phi) = \prod_{i=1}^{N} p(z_i \mid x_i,\phi) = \prod_{i=1}^{N}\prod_{j=1}^{D} \mathcal{N}(z_i \mid \mu_j(x_i),\sigma_j(x_i)).$$

Итого в нашей конструкции есть сразу две нейронных сети: первая получает на вход d-мерный вектор скрытых переменных z и выдает вектор размерности D, объект из распределения над x, а вторая получает на вход D-мерный вектор тренировочного примера x и выдает вектор размерности $2d$ с параметрами распределений на скрытые переменные z. Глубокая сеть может аппроксимировать крайне сложное, нелинейное распределение, то есть для вариационного приближения мы берем очень богатый класс распределений. Структура вариационного автокодировщика проиллюстрирована на рис. 10.3, *а*.

Остается только один вопрос — как же это все вместе обучить? Начнем с того, что запишем вариационную нижнюю оценку:

$$\mathcal{L}(q(Z \mid X,\phi),\boldsymbol{\theta}) = \mathcal{L}(\phi,\boldsymbol{\theta}) = \sum_{i=1}^{n} \int q(z_i \mid x_i,\phi) \log \frac{p(x_i,z_i \mid \boldsymbol{\theta})}{q(z_i \mid x_i,\phi)} \mathrm{d}z_i.$$

Эту функцию надо максимизировать по ϕ и $\boldsymbol{\theta}$. Как это сделать, если мы даже не можем подсчитать интеграл, то есть не можем даже подсчитать саму функцию, которую мы оптимизируем?

Чтобы оптимизировать, надо уметь считать или градиент, или хотя бы стохастический градиент. Обычный градиент считать сложно, уж функцию точно надо уметь считать. А вот стохастический — пожалуйста. Например, для функции вида $f(x) = \frac{1}{n}\sum_{i=1}^{n} f(x)$ стохастический градиент можно подсчитать, выбирая слагаемое случайным образом; при этом все становится в n раз быстрее, а стохастический градиент все равно будет несмещенной оценкой настоящего, то есть в ожидании получится то, что надо. Точно так же, если верно следующее:

$$f(x) = \mathbb{E}_{p(y)}[h(x,y))] = \int h(x,y)p(y)\mathrm{d}y,$$

то можно просто породить точку из распределения $p(y)$ и подсчитать производную $\frac{\partial h(x,y_0)}{\partial x}$, где точка y_0 порождена из распределения $p(y)$. И получится опять несмещенная оценка:

$$\int p(y \frac{\partial h(x,y)}{\partial x} \mathrm{d}y = \frac{\partial}{\partial x} \int p(y)h(x,y)\mathrm{d}y = \frac{\partial f}{\partial x}.$$

Можно, конечно, и получше оценить, с меньшей дисперсией, если взять выборку из нескольких точек и в качестве стохастического градиента усреднить получающиеся в этих точках производные. Если мы видим интеграл, представляющий математическое ожидание какой-то функции, можно заменить это ожидание на случайную выборку с соответствующим распределением.

А теперь чуть сложнее: пусть распределение в ожидании зависит от x:

$$f(x) = \int h(x,y)p(y \mid x)\mathrm{d}y.$$

Дифференцируем как произведение, что ж поделать:

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x} \int h(x,y)p(y \mid x)\mathrm{d}y = \int \left[\frac{\partial h(x,y)}{\partial x} p(y \mid x) + h(x,y)\frac{\partial p(y \mid x)}{\partial x}\right] \mathrm{d}y.$$

Первый интеграл можно теперь оценить точно так же, как выше — сгенерировать выборку, подставить частную производную, все хорошо. А вот со вторым теперь проблема: нигде нет математического ожидания.

Но здесь приходит на помощь так называемый log-derivative trick: если хочется представить $\frac{\partial p(y|x)}{\partial x}$ как $p(y \mid x)$, умноженное на что-то, то можно просто представить это в таком виде:

$$\frac{\partial p(y \mid x)}{\partial x} = p(y \mid x)\frac{\partial \log p(y \mid x)}{\partial x}.$$

Итого получается, что большой интеграл можно переписать так:

$$\int p(y \mid x)\left[\frac{\partial h(x,y)}{\partial x} + h(x,y)\frac{\partial \log p(y \mid x)}{\partial x}\right]\mathrm{d}y.$$

Подсчитать это честно, конечно, по-прежнему не получится, но нам же достаточно получить несмещенную оценку. Поэтому мы просто породим y из распределения $p(y \mid x)$ и подставим его туда:

$$\frac{\partial h(x,y_0)}{\partial x} + h(x,y_0)\frac{\partial \log p(y_0 \mid x)}{\partial x}, \quad \text{где} \quad y_0 \sim p(y \mid x).$$

Опять же, можно породить несколько точек и усреднить результаты. Заметьте, что мы ни разу нигде не брали интеграл, все, что нужно уметь делать — это порождать выборку из $p(y \mid x)$.

Теперь возвращаемся: нам нужно уметь максимизировать $\mathcal{L}(\phi,\boldsymbol{\theta})$ по ϕ и $\boldsymbol{\theta}$ по отдельности. Максимизация по $\boldsymbol{\theta}$ — это простой случай стохастического градиента, при фиксированных ϕ получится просто конкретное распределение $p(z_i \mid x_i,\phi)$, и из него можно породить выборку:

$$\frac{\partial}{\partial\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta},\phi) = \frac{\partial}{\partial\boldsymbol{\theta}}\sum_{i=1}^{n}\int q(z_i \mid x_i,\phi)\log\frac{p(x_i,z_i \mid \boldsymbol{\theta})}{q(z_i \mid x_i,\phi)}\mathrm{d}z_i \approx n\frac{\partial}{\partial\boldsymbol{\theta}}\log p(x_i,z \mid \boldsymbol{\theta}).$$

Приближенное равенство в этой формуле объясняется так: мы равномерно взяли случайную точку x_i из выборки и породили для нее $z \sim p(z \mid x_i,\phi)$.

А стохастический градиент по ϕ получается так, как описано выше, с помощью log-derivative trick. Важная проблема здесь в том, что у таких стохастических градиентов получается очень большая дисперсия. Есть большая наука о том, как понизить дисперсию в таких вычислениях, но в общем случае это пока открытая проблема, получаются только эвристические методы.

Но авторы вариационного автокодировщика придумали еще один трюк: репараметризацию (reparametrization trick)! Идея очень простая: если у нас есть выражение вида $\int f(x)p(x \mid \boldsymbol{\theta})\mathrm{d}x$, и мы хотим его продифференцировать по $\boldsymbol{\theta}$, давайте устраним параметры из распределения заменой переменных:

$$\frac{\partial}{\partial\boldsymbol{\theta}}\int f(x)p(x \mid \boldsymbol{\theta})\mathrm{d}x = \quad [p(x \mid \boldsymbol{\theta})\mathrm{d}x = p(\epsilon)\mathrm{d}\epsilon,\ x = g(\epsilon,\boldsymbol{\theta})] \quad =$$
$$= \frac{\partial}{\partial\boldsymbol{\theta}}\int f(g(\epsilon,\boldsymbol{\theta}))p(\epsilon)\mathrm{d}\epsilon \approx \frac{\partial}{\partial\boldsymbol{\theta}}f(g(\epsilon_0,\boldsymbol{\theta})),$$

где $\epsilon_0 \sim p(\epsilon)$.

Такой трюк возможен не всегда, но в нашем случае, когда распределения параметризованы нейронными сетями, он всегда проходит. Поэтому можно получить выражение для градиента по ϕ:

$$\frac{\partial}{\partial\phi}\mathcal{L}(\boldsymbol{\theta},\phi) = \frac{\partial}{\partial\phi}\sum_{i=1}^{n}\int q(z_i \mid x_i,\phi)\log\frac{p(x_i,z_i \mid \boldsymbol{\theta})}{q(z_i \mid x_i,\phi)}\mathrm{d}z_i =$$

$$[q(z_i \mid x_i,\phi)\mathrm{d}z_i = g(x_i,\epsilon) = \mathcal{N}(\epsilon \mid 0, I)\mathrm{d}\epsilon, \quad z_i = \mu(x_i) + \sigma(x_i)\cdot\epsilon]$$

$$= \frac{\partial}{\partial\phi}\sum_{i=1} n\int g(x_i,\epsilon)\log\frac{p(x_i,g(x_i,\epsilon) \mid \boldsymbol{\theta})}{q(g(x_i,\epsilon) \mid x_i,\phi)}\mathrm{d}\epsilon \approx n\frac{\partial}{\partial\phi}\log\frac{p(x_i,g(x_i,\epsilon) \mid \boldsymbol{\theta})}{q(g(x_i,\epsilon) \mid x_i,\phi)},$$

и теперь последнее выражение уже вполне можно и подсчитать, и дифференцировать, потому что это просто выходы наших двух нейронных сетей. Получающаяся структура графа вычислений изображена на рис. 10.3, *б*. Любопытно, что репараметризацию придумали почти одновременно сразу три группы исследователей — очевидно, эта идея действительно назревала.

На самом деле, хотя технических трудностей было немало, концептуально все довольно просто: мы ввели латентные переменные, попытались приблизить распределение q с помощью нейронных сетей (двух: одна по z генерирует x с параметрами $\boldsymbol{\theta}$, другая по x генерирует z с параметрами ϕ), и тем самым свели задачу функциональной оптимизации к параметрической. Дальше дело техники: как подсчитать стохастический градиент; получилась крайне эффективно обучаемая конструкция, и практические результаты ее тоже впечатляют. В исходной работе приводятся результаты на MNIST и датасете из разных выражений лица одного и того же человека. Имея это внутреннее представление, полученное вариационным автокодировщиком, мы можем решить, например, такую задачу: попробовать сгенерировать другие объекты, похожие на данный. Похожие в том самом многообразии низкой размерности, многообразии скрытых факторов.

Следующий шаг: если мы уже знаем структуру датасета, можно усложнить априорное распределение. Например, для распознавания цифр можно просто заменить нормальное распределение латентных переменных на смесь десяти гауссианов, в итоге обучится по одному гауссиану на циферку. Можно получить глубокое обобщение фильтра Калмана, скрытой марковской модели. Все это делается ровно так же, стохастические градиенты берутся по одному и тому же принципу.

И, как уже повелось в нашей книге, давайте приведем практический пример — это упрощенный и адаптированный пример из кода Яна Хендрика Метцена, опубликованного в 2015 году [358]. Мы будем, как и во многих других примерах в этой книге, использовать стандартный набор данных MNIST с рукописными цифрами. Начнем с импорта TensorFlow и загрузки датасета:

```
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
n_samples = mnist.train.num_examples
```

Затем проведем случайную инициализацию весов. Для этого будем использовать привычную нам инициализацию Ксавье (вспомните раздел 4.2). Для кодировщика и декодировщика возьмем по два уровня с 500 нейронами на каждом, размерность входа у MNIST равна $28 \times 28 = 784$, а размерность скрытого латентного пространства — 20:

```
def xavier_init(fan_in, fan_out, constant=1):
    low = -constant*np.sqrt(6.0/(fan_in + fan_out))
    high = constant*np.sqrt(6.0/(fan_in + fan_out))
    return tf.random_uniform((fan_in, fan_out),
                             minval=low, maxval=high,
                             dtype=tf.float32)


w, n_input, n_z = {}, 784, 20
n_hidden_recog_1, n_hidden_recog_2 = 500, 500
n_hidden_gener_1, n_hidden_gener_2 = 500, 500
w['w_recog'] = {
    'h1': tf.Variable(xavier_init(n_input, n_hidden_recog_1)),
    'h2': tf.Variable(xavier_init(n_hidden_recog_1, n_hidden_recog_2)),
    'out_mean': tf.Variable(xavier_init(n_hidden_recog_2, n_z)),
    'out_log_sigma': tf.Variable(xavier_init(n_hidden_recog_2, n_z))}
w['b_recog'] = {
    'b1': tf.Variable(tf.zeros([n_hidden_recog_1], dtype=tf.float32)),
    'b2': tf.Variable(tf.zeros([n_hidden_recog_2], dtype=tf.float32)),
    'out_mean': tf.Variable(tf.zeros([n_z], dtype=tf.float32)),
    'out_log_sigma': tf.Variable(tf.zeros([n_z], dtype=tf.float32))}
w['w_gener'] = {
    'h1': tf.Variable(xavier_init(n_z, n_hidden_gener_1)),
    'h2': tf.Variable(xavier_init(n_hidden_gener_1, n_hidden_gener_2)),
    'out_mean': tf.Variable(xavier_init(n_hidden_gener_2, n_input)),
    'out_log_sigma': tf.Variable(xavier_init(n_hidden_gener_2, n_input))}
w['b_gener'] = {
    'b1': tf.Variable(tf.zeros([n_hidden_gener_1], dtype=tf.float32)),
    'b2': tf.Variable(tf.zeros([n_hidden_gener_2], dtype=tf.float32)),
    'out_mean': tf.Variable(tf.zeros([n_input], dtype=tf.float32)),
    'out_log_sigma': tf.Variable(tf.zeros([n_input], dtype=tf.float32))}
```

Зададим скорость обучения, размер мини-батча и входную переменную:

```
l_rate=0.001
batch_size=100
x = tf.placeholder(tf.float32, [None, n_input])
```

Теперь все готово для того, чтобы реализовывать кодировщики и декодировщики. Кодировщик (распознаватель) берет входы и отображает их в нормальное распределение в пространстве скрытых признаков:

```
enc_layer_1 = tf.nn.softplus(tf.add(
    tf.matmul(x, w["w_recog"]['h1']), w["b_recog"]['b1']))
enc_layer_2 = tf.nn.softplus(tf.add(
    tf.matmul(enc_layer_1, w["w_recog"]['h2']), w["b_recog"]['b2']))
z_mean = tf.add( tf.matmul(
    enc_layer_2, w["w_recog"]['out_mean']),
    w["b_recog"]['out_mean'])
z_log_sigma_sq = tf.add( tf.matmul(
    enc_layer_2, w["w_recog"]['out_log_sigma']), w["b_recog"]['out_log_sigma'])
```

Затем мы набрасываем одну точку из нормального распределения в пространстве латентных признаков; чтобы породить точку из нормального распределения с заданными параметрами, достаточно набросить ее из стандартного нормального распределения (функцией `tf.random_normal`), а затем сдвинуть куда надо:

```
eps = tf.random_normal((batch_size, n_z), 0, 1, dtype=tf.float32)
z = tf.add(z_mean, tf.mul(tf.sqrt(tf.exp(z_log_sigma_sq)), eps))
```

Декодировщик (генератор) отображает точки в латентном пространстве в распределение Бернулли в пространстве данных:

```
dec_layer_1 = tf.nn.softplus(tf.add(
    tf.matmul(z, w["w_gener"]['h1']), w["b_gener"]['b1']))
dec_layer_2 = tf.nn.softplus(tf.add(
    tf.matmul(dec_layer_1, w["w_gener"]['h2']), w["b_gener"]['b2']))

x_reconstr_mean = tf.nn.sigmoid(tf.add(
    tf.matmul(dec_layer_2, w["w_gener"]['out_mean']),
    w["b_gener"]['out_mean']))
```

И теперь осталось только определить функцию потерь. Она состоит из двух частей. Первая — собственно ошибка восстановления, то есть минус логарифм правдоподобия входа в восстановленном распределении Бернулли (как обычно, мы добавляем маленькую константу, чтобы не пришлось считать логарифм нуля):

```
reconstr_loss = -tf.reduce_sum(x * tf.log(1e-10 + x_reconstr_mean) +
    (1-x) * tf.log(1e-10 + 1 - x_reconstr_mean), 1)
```

Вторая часть — это ошибка латентного распределения, которая определяется как расстояние Кульбака — Лейблера между распределением в пространстве латентных признаков, индуцированном кодировщиком на данных, и некоторым фиксированным априорным распределением, например обычным гауссианом:

```
latent_loss = -0.5 * tf.reduce_sum(1 + z_log_sigma_sq
    - tf.square(z_mean) - tf.exp(z_log_sigma_sq), 1)
cost = tf.reduce_mean(reconstr_loss + latent_loss)
```

Ошибка латентного распределения делает его максимально похожим на то самое априорное распределение (в нашем случае — стандартное нормальное распределение с нулевым средним и единичной дисперсией). Осталось только определить собственно оптимизатор; в этом качестве мы используем адаптивный алгоритм Adam (см. раздел 4.5):

```
optimizer = tf.train.AdamOptimizer(learning_rate=l_rate).minimize(cost)
```

Теперь можно обучать:

```
def train(sess, batch_size=100, training_epochs=10, display_step=5):
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(n_samples / batch_size)
        # Цикл по мини-батчам
        for i in range(total_batch):
            xs, _ = mnist.train.next_batch(batch_size)

            # Обучаем на текущем мини-батче
            _, c = sess.run((optimizer, cost), feed_dict={x: xs})
            # Compute average loss
            avg_cost += c / n_samples * batch_size

        # Каждые display_step шагов выводим текущую функцию потерь
        if epoch % display_step == 0:
            print("Epoch: %04d\tcost: %.9f" % (epoch+1, avg_cost))

init = tf.initialize_global_variables()
sess = tf.InteractiveSession()
sess.run(init)

train(sess, training_epochs=200, batch_size=batch_size)
```

После того как этот код отработает, мы получим автокодировщик, который понимает структуру возможных входов (рукописных цифр) и может как реконструировать цифры, так и порождать новые, выбирая скрытые факторы из нормального распределения. Чтобы посмотреть, как он реконструирует, достаточно подставить какие-нибудь новые рукописные цифры в качестве входов `x` и посмотреть, что из него получится в среднем реконструированного распределения `x_reconstr_mean`:

```
x_sample = mnist.test.next_batch(100)[0]
x_logits = sess.run(x_reconstr_mean_logits,
                    feed_dict={x: x_sample, eps:
                       np.random.normal(loc=0., scale=1., size=(batch_size, n_z))})
gen_logits = sess.run(x_reconstr_mean_logits,
                    feed_dict={z_mean:
                       np.random.normal(loc=0., scale=1., size=(batch_size, n_z))})
```

Результаты реконструкции показаны на рис. 10.4.

а *б* *в*

Рис. 10.4. Пример работы вариационного автокодировщика: *а* — исходные изображения; *б* — реконструированные; *в* — сэмплированные

А еще мы можем попробовать проверить, действительно ли в пространстве латентных признаков получается обещанное нормальное распределение. Давайте переобучим ту же модель, но с двумерным пространством латентных признаков, чтобы его можно было нарисовать и что-то понять в нем; для этого нужно просто в коде выше поменять `n_z=20` на `n_z=2`. Когда мы это сделаем, можно будет посмотреть, как цифры из тестового набора распределились в пространстве латентных признаков:

```
xs, ys = mnist.test.next_batch(5000)
z_mu = sess.run(z_mean, feed_dict={x: xs})
plt.figure(figsize=(8, 6))
plt.scatter(z_mu[:, 0], z_mu[:, 1], c=np.argmax(y_sample, 1))
plt.colorbar()
```

Результаты для трех разных этапов обучения показаны на рис. 10.5; на графиках точки, соответствующие разным цифрам, показаны разными маркерами. Действительно, общее распределение всех цифр во всех трех случаях выглядит весьма похожим на стандартное нормальное распределение. Но если до начала обучения все цифры в этом распределении перемешаны абсолютно случайным образом, то со временем в хаосе начинает появляться структура. К концу обучения уже довольно просто различить кластеры цифр в пространстве признаков, и при этом общая картинка все равно остается похожа на стандартный двумерный гауссиан! Учитывая, что мы взяли всего лишь двумерное пространство скрытых факторов, то есть существенно упростили модель, это отличный результат.

Можно пойти дальше: нарисовать конкретные примеры тех цифр, которые получаются из разных областей пространства латентных признаков. Для этого выберем точки в узлах решетки и построим соответствующую решетку из цифр:

```
nx = ny = 20
x_values = np.linspace(-3, 3, nx)
y_values = np.linspace(-3, 3, ny)

canvas = np.empty((28*ny, 28*nx))
for i, yi in enumerate(x_values):
    for j, xi in enumerate(y_values):
        z_mu = np.array([[xi, yi]])
```

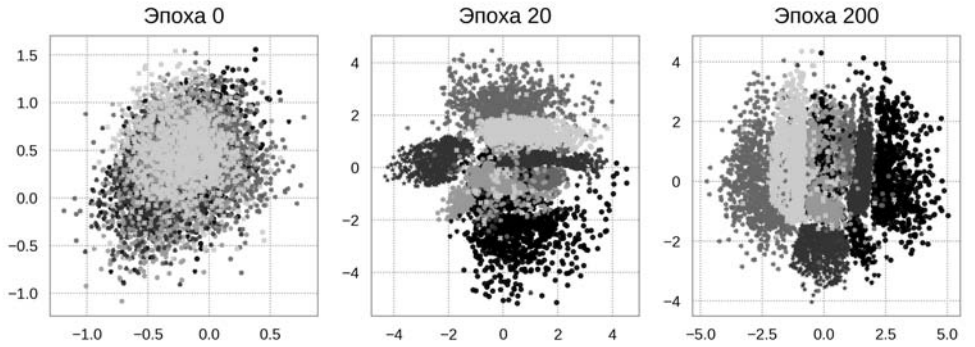


Рис. 10.5. 2D-представление распределения рукописных цифр в датасете MNIST

```
        x_mean = sess.run(x_reconstr_mean, feed_dict={z: z_mu})
        canvas[(nx-i-1)*28:(nx-i)*28, j*28:(j+1)*28] = x_mean[0].reshape(28, 28)

plt.figure(figsize=(8, 10))
Xi, Yi = np.meshgrid(x_values, y_values)
plt.imshow(canvas, origin="upper")
plt.tight_layout()
```

Результат показан на рис. 10.6: видно, что цифры меняются достаточно плавно, но при этом все время остаются вполне распознаваемыми, практически без непонятных промежуточных значений. Это и есть тот эффект, которого мы ожидаем от хорошего автокодировщика: путешествие через пространство латентных признаков не приведет нас к примерам, которых не бывает в реальной жизни, все время будут порождаться нормальные рукописные цифры (естественно, разные в разных областях пространства). Сравните, кстати, рис. 10.6 и результат 200 эпох обучения на рис. 10.5: эти картинки соответствуют друг другу, и вы видите соответствие между областями, приводящими к одним и тем же цифрам.

Давайте теперь сделаем следующий шаг и доведем наш пример до условного вариационного автокодировщика (conditional VAE). Эта конструкция аналогична условному состязательному автокодировщику (см. раздел 8.5): условный вариационный автокодировщик явным образом получает на вход метки цифр и в результате должен суметь порождать заданную цифру. Для этого нужно подать на вход кодировщику и декодировщику одну и ту же метку текущей цифры. В коде выше это можно сделать с совсем небольшими изменениями. Зададим заглушку для меток `y`, объединим векторы `x` и `y` и подадим оба на вход первого слоя кодировщика:

```
y = tf.placeholder(tf.float32, [None, 10])
xy = tf.concat([x, y], 1)
enc_layer_1 = tf.nn.tanh(tf.add(
       tf.matmul(xy, w["w_recog"]['h1']), w["b_recog"]['b1']))
```

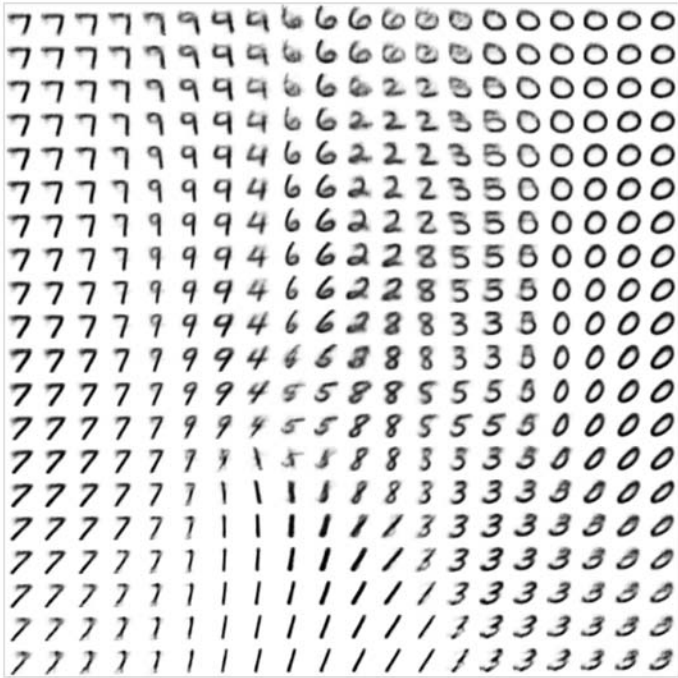


Рис. 10.6. Примеры рукописных цифр, сэмплированные из разных частей распределения скрытых факторов

Выход из первого слоя теперь тоже дополняется меткой текущего значения, и они вместе подаются в декодировщик:

```
zy_mean = tf.concat([z_mean, y], 1)
zy = tf.concat([z, y], 1)
dec_layer_1 = tf.nn.tanh(tf.add(
        tf.matmul(zy, w["w_gener"]['h1']), w["b_gener"]['b1']))
mean_dec_layer_1 = tf.nn.tanh(tf.add(
        tf.matmul(zy_mean, w["w_gener"]['h1']), w["b_gener"]['b1']))
```

Все остальное — абсолютно без изменений: ошибка по-прежнему складывается из ошибки автокодировщика и ошибки на скрытом слое, только теперь получается, что распределение скрытых факторов сравнивается с нормальным «по отдельности» для каждого класса, и на выходе получается стандартное нормальное распределение в каждом классе (см. рис. 10.7 — на нем никакой разницы между разными классами не просматривается). Зато теперь можно сделать условное порождение: на рис. 10.8 показаны примеры результатов сэмплирования с разными метками-условиями. Здесь нет общей двумерной картинки, как на рис. 10.6, но по-прежнему «путешествия» через пространство скрытых факторов, которые показаны в каждой строке рис. 10.8, все время приводят к разумным порожденным цифрам.

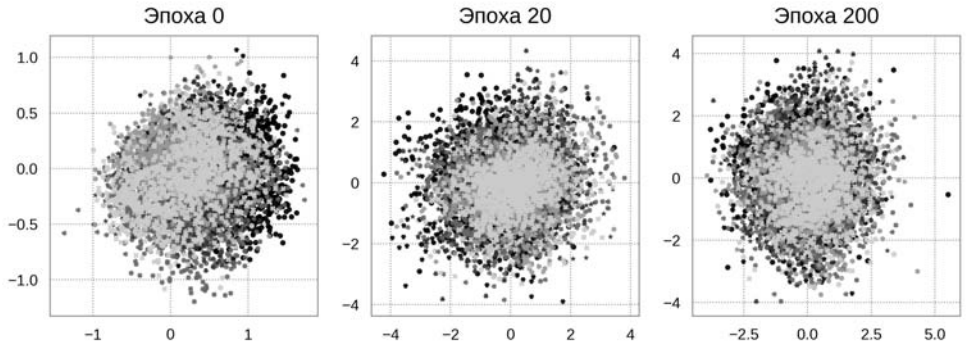


Рис. 10.7. 2D-представление распределения рукописных цифр в датасете MNIST в условном вариационном автокодировщике

Сейчас популярность вариационных автокодировщиков и разных вариантов этой архитектуры продолжает расти, они успешно соперничают с AAE за звание лучшей порождающей модели, основанной на нейронных сетях. Конечно, появились и расширения, и новые варианты вариационных автокодировщиков.

Например, в работе [550] строится конструкция так называемого вариационного автокодировщика с потерями (lossy VAE), в котором контроль над структурой скрытого представления используется для того, чтобы различить важные и неважные части поступающего на вход объекта (например, отделить объекты, изображенные на фото, от текстуры заднего плана), а затем «забыть» неважные части при дальнейшем порождении.

А в [394] строятся *непараметрические* вариационные автокодировщики, то есть VAE с непараметрическими априорными распределениями. Это значит, что они могут увеличивать сложность по мере усложнения и увеличения обучающего датасета, и эта идея далее развивается в *иерархические* непараметрические VAE, которые способны обучать иерархические структуры (фактически деревья) концепций и генерировать новые примеры из каждого узла получающихся деревьев.

Отдельно хочется отметить работу [400], которая предлагает общий взгляд на вариационные и состязательные автокодировщики. Оказывается, что и те, и другие в некотором смысле минимизируют дивергенцию Кульбака — Лейблера между истинным и модельным распределениями, просто в разных направлениях, и поэтому их можно рассматривать как две разные фазы классического алгоритма wake-sleep, который был разработан еще в 1990-е годы для обучения нейронных сетей без учителя [555], причем без участия Джеффри Хинтона и там не обошлось. Хотя мы уже не станем углубляться в детали этих и других современных расширений идей байесовских нейронных сетей, нет сомнений, что будущее глубокого обучения по крайней мере отчасти именно за такими методами.

Рис. 10.8. Примеры рукописных цифр, сэмплированные из разных частей распределения скрытых факторов условным вариационным автокодировщиком

10.5. Байесовские нейронные сети и дропаут

Лина. Ах, зачем вы меня схватили на руки и тащите к окну? Неужели вы хотите меня выбросить?

Ходотов. Нет, помилуйте... Это я вас так люблю!

А. Аверченко. На перепутье

А теперь вернемся к байесовскому выводу на собственно нейронных сетях, который мы анонсировали в первом разделе этой главы; такая последовательность изложения была нужна потому, что для вывода в *байесовских нейронных сетях* (Bayesian neural networks) мы опять будем активно использовать ту же самую идею вариационных приближений, которую объясняли в разделе 10.3 и затем расширяли аппроксиматорами в виде нейронных сетей. В качестве основных источников о современном байесовском выводе в нейронных сетях рекомендуем [171, 172, 191, 279] и недавно появившуюся диссертацию Ярина Гала [170].

Для нейронных сетей байесовский вывод в принципе выглядит точно так же, как и для любой другой вероятностной модели. Мы хотим найти апостериорное распределение:

$$p(\boldsymbol{w} \mid D) = \frac{p(D \mid \boldsymbol{w})p(\boldsymbol{w})}{p(D)} \propto p(D \mid \boldsymbol{w})p(\boldsymbol{w}),$$

где через $\boldsymbol{w}$ мы обозначили вектор всех весов сети, а через X — имеющиеся данные, а потом найти предсказательное распределение:

$$p(x \mid D) = \int_{\boldsymbol{w}} p(x \mid \boldsymbol{w}) p(\boldsymbol{w} \mid D) \mathrm{d}\boldsymbol{w} \propto \int_{\boldsymbol{w}} p(x \mid \boldsymbol{w}) p(D \mid \boldsymbol{w}) p(\boldsymbol{w}) \mathrm{d}\boldsymbol{w}.$$

В частности, в дальнейшем мы будем для определенности говорить о задаче классификации, когда $D = (X,Y)$ состоит из пар вида $(\boldsymbol{x}, y)$, и мы ищем апостериорное распределение:

$$p(\boldsymbol{w} \mid X, Y) = \frac{p(Y \mid X, \boldsymbol{w}) p(\boldsymbol{w})}{p(Y \mid X)} \propto p(Y \mid X, \boldsymbol{w}) p(\boldsymbol{w})$$

и предсказательное распределение:

$$p(y \mid \boldsymbol{x}, X, Y) = \int_{\boldsymbol{w}} p(y \mid \boldsymbol{x}, \boldsymbol{w}) p(\boldsymbol{w} \mid X, Y) \mathrm{d}\boldsymbol{w} \propto \int_{\boldsymbol{w}} p(y \mid \boldsymbol{x}, \boldsymbol{w}) p(Y \mid X, \boldsymbol{w}) p(\boldsymbol{w}) \mathrm{d}\boldsymbol{w}.$$

В нейронных сетях, разумеется, $p(\boldsymbol{w} \mid X, Y)$ имеет очень сложный вид, и аналитически найти это распределение не получается — нужны приближения.

Начнем с краткого исторического обзора. На протяжении всей книги мы раз за разом убеждались в том, что подавляющее большинство идей, использующихся в современных нейронных сетях, появились очень давно (по меркам машинного обучения, конечно): глубокие сети обучались с 1960-х годов, сверточные архитектуры известны и применяются на практике с конца 1980-х, LSTM разрабатывался в течение 1990-х и т. д. С байесовским выводом в нейронных сетях получилась ровно та же история: байесовские нейронные сети восходят к концу 1980-х годов [294, 534], а в явном виде идея добавить в нейронные сети априорные распределения и попробовать получать не одну оценку, а апостериорное распределение появилась в первой половине 1990-х, в работах известных специалистов по статистике и машинному обучению Рэдфорда Нила [383] и Дэвида Маккея [344]. Априорное распределение на веса обычно выбиралось нормальным, $p(\boldsymbol{w}) = \mathcal{N}(\mathbb{O}, \mathbb{I})$, а основная проблема состояла, как всегда в этом деле, в том, как провести вывод. В диссертации Нила [383] вывод был на основе методов Монте-Карло по схеме марковской цепи (Markov chain Monte Carlo, MCMC). А, например, в книге Кристофера Бишопа [44], которую мы всецело рекомендуем как одну из лучших книг о байесовском выводе, кульминацией главы о нейронных сетях становится описание основанного на лапласовском приближении алгоритма байесовского вывода, который предназначем для обучения параметров гауссовского приближения к апостериорному распределению. Таким образом можно получить приближенное апостериорное распределение, оптимизировать гиперпараметры и даже добраться до предсказательного распределения, заметно улучшив результаты, которые показывают «классические», неглубокие нейронные сети; но к современным глубоким сетям все эти рассуждения уже не вполне применимы.

Современный байесовский вывод в нейронных сетях основан, как мы уже упоминали, на вариационных приближениях. Как и в разделе 10.3, мы вводим новое распределение $q(\boldsymbol{w})$ и пытаемся приблизить им истинное апостериорное распределение $p(\boldsymbol{w} \mid X, Y)$, минимизируя расстояние Кульбака — Лейблера между ними:

$$\begin{aligned}
&\mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w} \mid X, Y)\right) = \\
&= \int q(\boldsymbol{w}) \log \frac{q(\boldsymbol{w})}{p(\boldsymbol{w} \mid X, Y)} \mathrm{d}\boldsymbol{w} = \int q(\boldsymbol{w}) \log \frac{q(\boldsymbol{w})}{p(\boldsymbol{w}) p(Y \mid X, \boldsymbol{w})} \mathrm{d}\boldsymbol{w} + \text{const} = \\
&= -\int q(\boldsymbol{w}) \log p(Y \mid X, \boldsymbol{w}) \mathrm{d}\boldsymbol{w} + \int q(\boldsymbol{w}) \log \frac{q(\boldsymbol{w})}{p(\boldsymbol{w})} \mathrm{d}\boldsymbol{w} + \text{const} = \\
&= -\int q(\boldsymbol{w}) \log p(Y \mid X, \boldsymbol{w}) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right) + \text{const} = \\
&= -\sum_{i=1}^{N} \int q(\boldsymbol{w}) \log p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right) + \text{const},
\end{aligned}$$

где в const спрятаны слагаемые, не зависящие от $q(\boldsymbol{w})$ (они не участвуют в минимизации), а в последней строке правдоподобие $p(Y \mid X, \boldsymbol{w})$ разложено в произведение по точкам данных.

Впервые такую минимизацию попытались провести, опять же, в первой половине 1990-х годов, в работе [226], где использовалось очень сильное предположение о форме приближения. Там предполагалось, что $q(\boldsymbol{w})$ полностью раскладывается в произведение нормальных распределений по отдельным весам:

$$q(\boldsymbol{w}) = \prod_{w \in \boldsymbol{w}} q_{\mu_w, \sigma_w}(w) = \prod_{w \in \boldsymbol{w}} \mathcal{N}(w \mid \mu_w, \sigma_w).$$

Даже это оказалось нелегко: только для сетей с одним скрытым уровнем результат удается получить аналитически, что и было проделано в [226]. Но в любом случае метод, не учитывающий корреляции между весами, дает не самые лучшие результаты, а попытки добавить приближению выразительности [31] приводят к тому, что алгоритмы вывода становятся квадратичными от числа весов в сети — для современных огромных нейронных сетей это смерти подобно.

Основных сложностей с исходной задачей минимизации две: во-первых, $\int q(\boldsymbol{w}) p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w}$ никак не вычислить, если у сети больше одного скрытого уровня, во-вторых, получается, что даже чтобы подсчитать функцию, которую мы минимизируем, нужно взять сумму по всему датасету. Вторую проблему решить не так уж сложно: давайте вместо суммирования по всем N тренировочным примерам выберем из них случайное подмножество S размера $M < N$ и перевзвесим. Теперь мы оптимизируем такую функцию:

$$\mathcal{L}(\boldsymbol{w}) = -\frac{N}{M} \sum_{i \in S} \int q(\boldsymbol{w}) p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right).$$

Если S выбирать случайно, получится несмещенная оценка исходной суммы. Примерно так же мы когда-то заменяли градиентный спуск (который тоже, формально говоря, требует суммирования по всему датасету) стохастическим градиентным спуском, где сумма берется по мини-батчу вместо всего датасета, а теперь мы выбираем мини-батч S для вычисления функции оптимизации.

С интегралом придется повозиться. Есть разные способы построить хорошую стохастическую оценку для этого интеграла [45, 405], но нам сейчас опять поможет предложенный в [279, 280] трюк, с которым мы уже познакомились в разделе 10.3.

Прежде всего заметим, что нам нужно оценить не сам интеграл, а производную от него, которую мы могли бы использовать в градиентном спуске:

$$I(\boldsymbol{\theta}) = \frac{\partial}{\partial \boldsymbol{\theta}} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x.$$

Предположим теперь (репараметризация), что $p_{\boldsymbol{\theta}}(x)$ можно параметризовать как $p(\epsilon)$ уже без параметров, где $x = g(\boldsymbol{\theta}, \epsilon)$. Это, например, верно для нормального распределения: если $p_{\boldsymbol{\theta}}(x) = \mathcal{N}(x \mid \mu,\sigma^2)$, то $g(\boldsymbol{\theta},\epsilon) = \mu + \sigma\epsilon$, и $p(\epsilon) = \mathcal{N}(\epsilon \mid \mathbb{O},\mathbb{I})$. Теперь, точно как выше, мы получаем несмещенную оценку:

$$\frac{\partial}{\partial \boldsymbol{\theta}} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x \approx \frac{\partial}{\partial \boldsymbol{\theta}} f(g(\boldsymbol{\theta},\epsilon)) = f'(g(\boldsymbol{\theta},\epsilon)) \frac{\partial}{\partial \boldsymbol{\theta}} g(\boldsymbol{\theta},\epsilon).$$

Например, для нормального распределения $x = \mu + \sigma\epsilon$, и мы получим:

$$\begin{aligned}
\frac{\partial}{\partial \mu} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x &= \int f'(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x, \\
\frac{\partial}{\partial \sigma} \int f(x) p_{\boldsymbol{\theta}}(x) \mathrm{d}x &= \int f'(x) \frac{x-\mu}{\sigma} p_{\boldsymbol{\theta}}(x) \mathrm{d}x.
\end{aligned}$$

И теперь можно подставить эту оценку обратно в целевую функцию $\mathcal{L}(\boldsymbol{w})$. Сделаем репараметризацию:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\theta}) = -\frac{N}{M} \sum_{i \in S} \int q_{\boldsymbol{\theta}}(\boldsymbol{w}) \log p(y_i \mid f_{\boldsymbol{w}}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{w} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right) = \\
= -\frac{N}{M} \sum_{i \in S} \int p(\boldsymbol{\epsilon}) \log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i)) \mathrm{d}\boldsymbol{\epsilon} + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right),
\end{aligned}$$

а затем подставим несмещенную стохастическую оценку вместо интеграла:

$$\hat{\mathcal{L}}(\boldsymbol{\theta}) = -\frac{N}{M} \sum_{i \in S} \log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i)) + \mathrm{KL}\left(q(\boldsymbol{w}) \| p(\boldsymbol{w})\right).$$

Таким образом, один шаг алгоритма минимизации расхождения между $q_{\boldsymbol{\theta}}(\boldsymbol{w})$ и $p(\boldsymbol{w} \mid X,Y)$ будет выглядеть так:

- сначала сэмплировать M случайных примеров из тренировочного набора размера N и M случайных величин $\hat{\boldsymbol{\epsilon}}_i \sim p(\boldsymbol{\epsilon})$;
- затем вычислить обновление весов:

$$\Delta\boldsymbol{\theta} = -\frac{N}{M}\sum_{i\in S}\frac{\partial}{\partial\boldsymbol{\theta}}\log p(y_i \mid f_{g(\boldsymbol{\theta},\hat{\boldsymbol{\epsilon}}_i)}(\boldsymbol{x}_i)) + \frac{\partial}{\partial\boldsymbol{\theta}}\mathrm{KL}\left(q(\boldsymbol{w})\|p(\boldsymbol{w})\right).$$

Этот алгоритм, постепенно обновляя параметры $\boldsymbol{\theta}$ распределения $q_{\boldsymbol{\theta}}(\boldsymbol{w})$, приведет это распределение как можно ближе к истинному апостериорному распределению $p(\boldsymbol{w} \mid X,Y)$ — ровно то, что мы и хотели сделать. Потом можно будет и байесовские предсказания делать обычным стохастическим методом: посэмплировать несколько разных наборов весов $\hat{\boldsymbol{w}}_r \sim q_{\boldsymbol{\theta}}(\boldsymbol{w})$ и усреднить результаты:

$$q_{\boldsymbol{\theta}}(y \mid \boldsymbol{x}) := \frac{1}{R}\sum_{r=1}^{R} p(y \mid \boldsymbol{x},\hat{\boldsymbol{w}}_r).$$

Но это еще не все — самое интересное только начинается! Давайте для примера рассмотрим обычную полносвязную нейронную сеть с одним скрытым слоем. У нее три вида параметров: матрица весов входного слоя W_1, матрица весов скрытого слоя W_2 и вектор свободных членов $\boldsymbol{b}$, то есть $\boldsymbol{\theta} = (W_1,W_2,\boldsymbol{b})$. Давайте выберем в качестве $q_{\boldsymbol{\theta}}(\boldsymbol{w})$ одно конкретное семейство распределений, которое выглядит так:

- сначала возьмем бинарные случайные величины $\boldsymbol{\epsilon}_1$ и $\boldsymbol{\epsilon}_2$, которые являются результатами независимых бросков нескольких монеток с вероятностями p_1 и p_2, и сэмплируем их, получив векторы $\hat{\boldsymbol{\epsilon}}_1$ и $\hat{\boldsymbol{\epsilon}}_2$;
- затем построим из них диагональные матрицы $\mathrm{diag}(\hat{\boldsymbol{\epsilon}}_1)$ и $\mathrm{diag}(\hat{\boldsymbol{\epsilon}}_2)$;
- и определим функцию $g(\boldsymbol{\theta},\hat{\boldsymbol{\epsilon}}) = (\mathrm{diag}(\hat{\boldsymbol{\epsilon}}_1)W_1, \mathrm{diag}(\hat{\boldsymbol{\epsilon}}_2)W_2, \boldsymbol{b})$.

При таком выборе функции g получится, что при вычислении целевой функции $\hat{\mathcal{L}}(\boldsymbol{\theta})$ для нашей минимизации мы считаем выход нейронной сети как будто с дропаутом с вероятностью p_1 после первого слоя и p_2 после второго! Если же мы теперь выберем в качестве априорного распределения $p(\boldsymbol{w})$ независимые нормальные распределения на каждом из весов, то целевая функция будет пропорциональна следующей величине:

$$-\frac{N}{M}\sum_{i\in S}\log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i)) + \lambda_1\|W_1\|^2 + \lambda_2\|W_2\|^2 + \lambda_3\|\boldsymbol{b}\|^2,$$

то есть мы получим, что обычный вывод в нейронных сетях с дропаутом в точности соответствует байесовскому выводу в вариационном приближении с таким семейством распределений q[1].

[1] Доказательства этого утверждения и того, что $\log p(y_i \mid f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}(\boldsymbol{x}_i))$ соответствует целевым функциям в обычных задачах регрессии и классификации, достаточно просты. Мы оставляем их читателю

Получилась вот какая последовательность. Дропаут изначально был инженерным решением, некоторое время это был просто трюк, который позволял нейронным сетям работать лучше. Конечно, люди и раньше пытались давать теоретические объяснения феномену дропаута. Например, Джеффри Хинтон в своих лекциях рассказывал о том, как дропаут представляет собой в некотором роде «байесовское усреднение» огромного числа разных моделей, отличающихся друг от друга архитектурой: все они делят между собой веса, и каждая модель обучается только на одном-единственном мини-батче, когда выпадает эта конкретная конфигурация выброшенных нейронов.

Некий образ связи между байесовским выводом и дропаутом в этом объяснении есть, но только после процитированных выше работ [171, 279], относящихся к 2015 году, дропауту наконец-то было дано действительно полное и удовлетворительное теоретическое обоснование[1].

Зачем нужно это обоснование? Казалось бы, дропаут и так отлично работает. Одно важное следствие состоит в том, что изначально доля выбрасываемых нейронов в дропауте была фиксированным числом. Но теперь, когда мы переформулировали обучение с дропаутом в виде максимизации вариационной нижней оценки, мы без проблем можем оптимизировать долю дропаута тоже: для этого достаточно просто зафиксировать остальные веса и максимизировать ту же самую оценку по доле дропаута! Более того, мы теперь можем подбирать ее индивидуально: для каждого слоя, для каждого нейрона, даже для каждой связи между нейронами. Этот метод получил название *вариационный дропаут* (variational dropout) [279]. Изначальная процедура дропаута таких вольностей совершенно не предусматривала, из нее не было понятно, как можно автоматически настраивать вероятность дропаута. При этом никакого страха переобучиться нет — мы не меняем априорное распределение, просто все точнее и точнее приближаем апостериорное. Более того, выяснилось, что вариационный дропаут добавляет разреженности в глубокие нейронные сети, то есть обнуляет подавляющее большинство весов [374]; этот результат, кстати, был получен в России, в лаборатории Дмитрия Ветрова.

А другое важное следствие состоит в том, что теперь мы можем лучше понять, как нужно делать дропаут в более сложных моделях. Рассмотрим, например, такой вопрос: как делать дропаут в рекуррентных сетях? В известной работе [583] проводилось подробное исследование методов дропаута для рекуррентных сетей. Авторы пришли к выводу, что делать дропаут нужно только между слоями, а между узлами одного рекуррентного слоя не нужно (рис. 10.9). Этому нашлось очень

в качестве упражнения, а также ссылаемся на [171, 279], где можно найти и много других интересных замечаний об этом выводе.

[1] Кстати, примерно такая же история случилась в конце 1990-х годов с другим известным аппаратом машинного обучения, бустингом: некоторое время он «просто работал», а потом уже удалось понять, что в бустинге происходит на самом деле и какой функционал оптимизируется. И сразу же оказалось, что когда исследователи поняли функционал для исходной конструкции бустинга, они сразу же смогли его обобщить, распространить на другие случаи и получить много новых моделей и алгоритмов вывода.

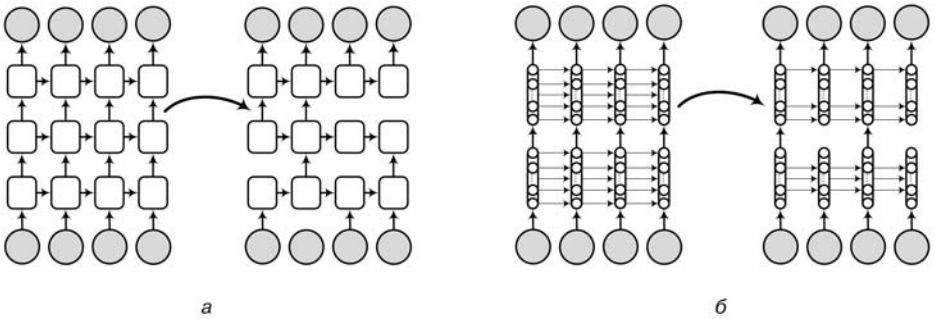


Рис. 10.9. Дропаут в рекуррентных сетях:
а — дропаут только между уровнями [583]; *б* — дропаут везде [169]

логичное объяснение: дропаут между узлами одного рекуррентного слоя разрушает возможность сделать долгосрочную память, ведь связь с предыдущим нейроном очень часто будет нарушена, а мы хотим как раз обучить как можно более долгосрочные зависимости. Однако после того как появилось теоретическое обоснование для дропаута, тут же родилась и идея о том, как его делать внутри рекуррентного слоя [169].

Действительно, давайте рассмотрим функцию $f_y = f_{g(\boldsymbol{\theta},\boldsymbol{\epsilon})}$ в случае рекуррентной сети (для простоты обычной, но на LSTM и GRU это тоже можно распространить). Она теперь зависит еще от скрытого состояния, оно, в свою очередь, зависит от весов на предыдущем уровне через функцию f_h, и так далее:

$$f_y(\boldsymbol{h}_t) = f_y(f_h(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})) = f_y(f_h(\boldsymbol{x}_t, f_h(\boldsymbol{x}_{t-1} f_h(\ldots f_h(\boldsymbol{x}_1, \boldsymbol{h}_0)\ldots)).$$

Мы знаем, что дропаут соответствует подбрасыванию монетки для каждого веса матрицы. Матриц, как мы помним из главы 6, у рекуррентной сети три: входная U, выходная V и матрица рекуррентных весов W. Интуиция, описанная в [583], говорит, что дропаут на матрице W не работает, потому что рекуррентные связи должны сохраняться. Поэтому дропаут должен выглядеть как на рис. 10.9, *а*, сохраняя все рекуррентные связи и выбрасывая часть связей между уровнями.

Но если добавить дропаут по схеме выше, в виде подбрасывания монетки *один раз* для каждого столбца матрицы W, то эта проблема исчезает: да, матрица W участвует в формуле много раз, но все связи во времени между нейронами будут сохраняться, потому что каждый нейрон будет либо включен, либо выключен одновременно на всем протяжении входной последовательности. Таким образом, получается, что дропаут на рекуррентных связях использовать можно, нужно только зафиксировать случайно выбрасываемые нейроны один раз для каждого входа (или мини-батча), а на разных входах можно бросать монетки заново, ничего

не испортится. Такой дропаут мы изобразили рис. 10.9, *б*: теперь можно выбрасывать не только связи между уровнями, но и отдельные компоненты рекуррентных связей, только делать это надо одинаково на протяжении всего слоя. В [169] приводятся результаты экспериментов, достаточно убедительно показывающие, что такой дропаут действительно делает рекуррентные сети лучше и предотвращает оверфиттинг без ухудшения качества.

Байесовские нейронные сети — это очень быстро развивающаяся область. Все эти результаты были получены буквально в течение последнего года-двух, но уже есть и конкретные практические применения. Например, в работе [488] строится система автоматического перевода на основе корпуса переводных новостей в рамках конкурса, проводившемся на конференции WMT 2016 [157] (кстати, среди построенных моделей есть и перевод с английского на русский и обратно). Авторы использовали стандартную схему с энкодером, декодером и сетью внимания (вспомним раздел 8.1), но в некоторых случаях оказалось, что для того чтобы избежать оверфиттинга, нужно было использовать дропаут в рекуррентной архитектуре. Они использовали вариационный дропаут по приведенной выше схеме и получили существенное улучшение. В работе [574] рассматривается задача «сшивания» вместе частей одного изображения, в частности для таких медицинских применений, как МРТ разных частей одного и того же мозга. Авторы построили модель, основанную на сверточных сетях, но рассмотрели ее с байесовских позиций. Дело в том, что разные изображения здесь «сшиваются» не вполне точно: между разными снимками проходит время, они могут быть в разной степени зашумлены, пациент мог немного сдвинуться и так далее. И для медицины важно понимать, какие части полученного снимка мы знаем точно, а какие не очень; для этого и хотелось бы знать все апостериорное распределение целиком и уметь оценивать его дисперсию, а не только получить одну оптимальную точку. Аналогичные применения, в которых у изображения появляется «карта неопределенности», оценивающая неопределенность полученной модели в разных его частях, появились и, например, при создании беспилотных автомобилей [276, 277].

Есть и другие интересные применения, но закончить хочется все-таки на том, что байесовские нейронные сети — это, несомненно, будущее обучения нейронных сетей. Уже сам факт того, что эмпирические «трюки» обучения глубоких сетей постепенно начинают получать строгие математические обоснования, вселяет надежду. Байесовские объяснения происходящего могут привести к целому ряду новых важных продвижений. Например (см. также [172]):

- если мы понимаем, как связаны методы обучения и регуляризации с априорными распределениями и алгоритмами приближенного вывода, мы можем получить новые методы обучения, просто подставляя другие априорные распределения, те, которые лучше подходят для конкретной задачи;
- байесовское обучение обычно можно продолжить до обучения гиперпараметров, то есть параметров априорных распределений, из которых берутся параметры модели;

- байесовские методы, как в упомянутых выше приложениях, позволяют оценивать неопределенность, остающуюся в модели после обучения; это тоже имеет очевидную ценность для приложений;
- многие классические байесовские модели достигают хороших результатов, используя очень простые базовые предположения (например, тематическое моделирование работает, превращая тексты в мешки слов); выразительность этих моделей наверняка можно значительно улучшить, заменив упрощающие предположения на более гибкие модели, формализуемые с помощью глубоких нейронных сетей.

Итак, как мы увидели на протяжении этой главы, байесовские модели и глубокие нейронные сети могут помогать друг другу: нейронные сети делают байесовские модели более гибкими и богатыми, а байесовское обучение позволяет лучше понимать, как обучать нейронные сети. Однако сейчас эта область находится только в самом начале пути и ждет новых исследователей. Дерзайте, коллеги!

10.6. Заключение: что не вошло в книгу и что будет дальше

> ...Она совсем не из этой страны — она из тех светозарных краев, где нас давным-давно ждут. Там вечно искрится роса и колышется стройный тростник.
>
> *Г. Миллер. Тропик Козерога*

Вот и подходит к концу наша книга. В заключение мы расскажем о нескольких направлениях, которые в книгу не вошли — как знать, может быть, они будут ждать читателей в следующих изданиях — а также попробуем дать прогноз о том, куда будет дальше двигаться человеческая мысль.

В главе 5 мы подробно говорили о современных архитектурах для распознавания изображений, которые обучаются на ImageNet, то есть классифицируют фотографию согласно тому, какой объект на ней изображен. Однако в жизни мы, когда смотрим вокруг себя, решаем не эту задачу, а задачу *распознавания объектов* (object detection): на фотографии может быть много чего изображено, и мы умеем распознавать каждый из этих объектов и указывать, где именно он находится. А иногда приходится решать и задачу *сегментации* (segmentation): не просто распознать, какие и сколько объектов на фото, но и буквально уметь указывать, какие пиксели фотографии им принадлежат. Для этого применяются архитектуры, которые основаны на все тех же VGG или ResNet, но имеют дополнительные слои, которые обучаются отвечать на вопрос, где именно находятся на фотографии те или иные объекты. Лидерами этой области сейчас являются такие модели, как YoLo [444, 576] и Faster R-CNN [152] для распознавания объектов и SegNet [20], U-Net [454] и Mask R-CNN [353] для сегментации.

В главе 8 мы достаточно кратко поговорили о моделях с вниманием и обошли тем самым вниманием несколько интересных направлений развития этой идеи. Так, для ответов на вопросы применяются так называемые *сети с памятью* (memory networks) [10, 561, 565], которые могут явным образом хранить вход и управлять этой памятью с помощью своеобразного механизма внимания, который решает, к какой именно части памяти нужно сейчас обратиться.

А следующий шаг в развитии этой идеи сделали так называемые *нейронные машины Тьюринга* (neural Turing machines) [194], которые используют своеобразный вариант механизма внимания для того, чтобы буквально управлять лентой машины Тьюринга и обучаться тем или иным алгоритмам по набору входов и выходов. Современные «нейрокомпьютеры» обучаются с помощью обучения с подкреплением [582], и сейчас это направление активно развивается во все той же компании *DeepMind*. Последний вариант такой модели, получивший громкое (но заслуженное) название *дифференцируемый нейронный компьютер* (differentiable neural computer, DNC), способен уже управлять практически настоящей «оперативной памятью», делая чтение и запись по адресам, и решать задачи, придумывая алгоритмы на графах [240].

Порождающие состязательные сети (глава 8), глубокое обучение с подкреплением (глава 9) и нейробайесовское обучение, которому посвящена эта глава, сейчас развиваются так быстро, что для книги глупо и пытаться оставаться на переднем крае. Очередной набор важных новых идей был представлен на конференции ICML в августе 2017 года. Например, в работе [450] представлен вариант обучения с подкреплением с противником (adversarial reinforcement learning), в котором противник активно пытается сбить агента с толку теми или иными дестабилизирующими действиями. В [8] разработан вариант GAN, в котором вместо дивергенции Йенсена — Шеннона используется расстояние Вассерштейна (которое обычно называется Earth Mover's Distance, EMD), что приводит к более стабильному обучению и меньшему схлопыванию мод. А работа [357] развивает новый метод обучения вариационных автокодировщиков, который... представляет максимизацию правдоподобия как игру между двумя игроками и основан на соперничающих сетях.

В последнее время наблюдается очевидный тренд на сближение между состязательными сетями и байесовскими методами: в работе [461] предлагается конструкция байесовского варианта GAN, а в [549] к GAN успешно применяются методы вариационного вывода. Мы полагаем, что ближайшее будущее — именно за такой «смычкой» между байесовскими и состязательными методами.

Есть и новые повороты более классических идей: например, в [440] предлагаются так называемые recurrent highway networks (да, в этой работе тоже не обошлось без Юргена Шмидхубера), которые расширяют архитектуру LSTM новыми связями и тем самым радикально улучшают, например, языковое моделирование. И это лишь несколько примеров. В целом, сейчас можно смело открывать список работ любой ведущей конференции по машинному обучению (две самые лучшие — это

ICML, International Conference on Machine Learning, и NIPS, Annual Conference on Neural Information Processing Systems), обязательно будет очень много интересного, а если вы прочли эту книгу, то и с большинством современных статей по обучению глубоких сетей должны справиться.

И, наконец, главный вопрос: что будет дальше? Куда все это катится? Честно скажем, что пытаться детально предсказывать развитие науки — занятие абсолютно бесперспективное. Если бы мы знали, что будет дальше, мы бы уже давно это сделали. Но некий общий вектор увидеть можно.

Общее настроение в начале революции глубокого обучения было, если можно так выразиться, «луддистским». Внезапно оказалось, что вероятностные модели, которые составляли основное содержание машинного обучения в течение почти двух десятилетий, как бы и «не нужны»: можно просто придумать удачную архитектуру нейронной сети, начать обучать ее градиентным спуском, и при достаточно большом датасете и достаточно мощной видеокарте все обязательно получится. Возможно, вы и сами вынесли такое ощущение из первых глав этой книги: мы не раз подчеркивали, что многие прорывы в машинном обучении, о которых мы писали ранее, получены «всего лишь градиентным спуском», безо всякой особенно сложной математики, хоть и не без важных и неочевидных трюков. Но сейчас постепенно становится понятным, что без математики все-таки не обойтись. В этой главе мы попытались дать очень краткое введение в то, что сейчас происходит на переднем крае обучения глубоких сетей. Хотя сейчас революция глубокого обучения все еще в самом разгаре, и в этой науке расцветают все цветы, мы полагаем, что будущее — за байесовскими методами и слиянием вероятностных моделей с нейронными сетями. Именно на этом пути получены самые интересные из недавних результатов, и именно в этом направлении двигаются ведущие исследователи. В результате глубокие нейронные сети будут делать то, что они делают лучше всего: служить универсальными аппроксиматорами для очень сложных функций; а функции эти будут, например, плотностями интересующих нас распределений. Как это всегда и бывает, очередной виток спирали развития науки не отменяет предыдущий, а расширяет, усложняет и улучшает его.

Еще одно большое направление, которое сейчас только начинается, состоит в том, чтобы объединять разных «агентов», выраженных нейронными сетями (а возможно, и другими типами моделей машинного обучения) в некую единую архитектуру, которая могла бы использовать разных агентов для различных задач и не переобучаться для каждого нового типа задач заново. Теоретически, именно это должен так или иначе делать искусственный интеллект общего назначения, который мог бы иметь шансы достичь уровня человека.

Интересный первый шаг в этом направлении — модель PathNet, представленная все тем же DeepMind [414]. PathNet — это модулярная нейронная сеть, состоящая из нескольких уровней, на каждом из которых расположено несколько «модулей», каждый из которых тоже является нейронной сетью. Для разных задач PathNet строит новые пути через эту модулярную архитектуру, то есть для задачи

распознавания рукописных цифр может быть использовано другое подмножество модулей, чем для задачи игры в приставку Atari, но модули при этом общие, а разные пути выбираются по сути эволюционным методом, генетическим алгоритмом. Результаты в [414] достаточно убедительны, и хотя пока это только первый шаг, нет сомнений, что перенос обучения (transfer learning) и построение архитектур общего назначения — это одна из центральных задач в машинном обучении на ближайшие годы.

А закончить хочется результатами опросов, которые в 2014 году были подытожены в работе Винсента Мюллера и уже упоминавшегося нами Ника Бострома [380]. Они опрашивали ведущих экспертов в области искусственного интеллекта; в опросе было несколько групп участников, от участников конференции AGI (Artificial General Intelligence) до топ-100 ученых в этой области по индексу цитирования (кстати, ответили 29 из 100, что показывает серьезность затеи). На вопрос о том, когда мы сможем разработать полноценный искусственный интеллект человеческого уровня, медианный ответ был — к 2075 году (интересно, что топ-100 ученых здесь были даже чуть более оптимистичны, с медианой в 2070 году), а вероятность в 50 % этому событию половина участников опроса дают уже в 2040 году или раньше.

Это значит, что вовсе не исключено (хотя, конечно, никем и не гарантировано), что мы с вами, дорогие читатели, доживем до того момента, когда искусственный интеллект сравняется с нами... а там и превзойдет. После достижения паритета искусственному интеллекту уже вряд ли что-то сильно помешает нас превзойти. И тогда наша жизнь неизбежно изменится до полной неузнаваемости: даже если не будет никаких «взрывов» и «сингулярностей», скорость технологического и научного прогресса наверняка вырастет настолько, что все современные жалобы на «постоянно меняющийся мир», которые так любят некоторые психологи, покажутся детским лепетом. Как ужиться с настоящим искусственным интеллектом, а не просто набором специализированных моделей, каждая из которых умеет только играть в го или водить машину, — это вопрос, который нам с вами, возможно, придется решать в течение нашей жизни. И от того, как мы его решим, зависит все будущее нашей с вами цивилизации.

А может, все будет совсем не так. И это ужасно интересно.

Благодарности

Мы очень благодарны нашим коллегам, которые прочитали рукопись и высказали замечания, позволившие нам сделать эту книгу лучше. Это были:

- Валентин Малых, исследователь лаборатории нейронных систем и глубокого обучения МФТИ;
- Елена Тутубалина, исследователь кафедры интеллектуальных технологий поиска Казанского (Приволжского) федерального университета;
- Павел Нестеров, энтузиаст машинного обучения, активист OpenDataScience, движения, собирающего вместе лучших русскоязычных специалистов по машинному обучению и анализу данных.

Сергей Николенко благодарен своим коллегам и аспирантам из Санкт-Петербургского отделения Математического института им. В. А. Стеклова РАН, с которыми уже более десяти лет ведет плодотворное сотрудничество (особенно Эдуарду Гиршу, Дмитрию Ицыксону и Александру Куликову), коллегам и студентам из Высшей школы экономики в Санкт-Петербурге (особенно Александру Сироткину), коллегам из компании Neuromation (особенно Максиму Прасолову и Константину Гольцеву), а также лично Кириллу Когану (IMDEA, Мадрид). Работать со всеми вами — одно удовольствие, и я надеюсь, что наше сотрудничество будет продолжаться и впредь.

Артур Кадурин благодарен сыну Михаилу и жене Александре, которые часто позволяли ему жертвовать семейными обязательствами ради написания этой книги. Коллегам и друзьям — Ивану Баскову, Павлу Нестерову, Кузьме Храброву — активно следившим за процессом и мотивирующим на продолжение. А также бывшему руководителю — Михаилу Фирулику, который помог встать на путь датасайентиста и сделать на этом пути первые шаги.

Екатерина Архангельская благодарна своей семье, а в особенности своему дедушке Валентину Михайловичу за поддержку и понимание в процессе написания этой книги.

Литература

1. *Abdi H., Williams L. J.* Principal Component Analysis // Wiley Interdisciplinary Reviews: Computational Statistics, 2010, vol. 2, no. 4. — P. 433–459.
2. *Ackley D. H., Hinton G. E., Sejnowski T. J.* A Learning Algorithm for Boltzmann Machines // Connectionist Models and Their Implications: Readings from Cognitive Science / Norwood, NJ, USA: Ablex Publishing Corp., 1988. — P. 285–307.
3. Addressing the Rare Word Problem in Neural Machine Translation / T. Luong et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 11–19.
4. Adversarial Autoencoders / A. Makhzani et al. // arXiv, 2015. http://arxiv.org/abs/1511.05644.
5. *Aggarwal C. C., Reddy C. K.* Data Clustering: Algorithms and Applications, Chapman and Hall/CRC, 2013.
6. *Antipov G., Baccouche M., Dugelay J.* Face Aging With Conditional Generative Adversarial Networks // arXiv, 2017. http://arxiv.org/abs/1702.01983.
7. Application Of Pretrained Deep Neural Networks To Large Vocabulary Speech Recognition / N. Jaitly et al. // Proceedings of Interspeech 2012, 2012.
8. *Arjovsky M., Chintala S., Bottou L.* Wasserstein Generative Adversarial Networks // Proc. 34th ICML / vol. 70 of *Proceedings of Machine Learning Research*, PMLR, 2017. — P. 214–223.
9. *Arjovsky M., Shah A., Bengio Y.* Unitary Evolution Recurrent Neural Networks // arXiv, 2015. http://arxiv.org/abs/1511.06464.
10. Ask Me Anything: Dynamic Memory Networks for Natural Language Processing / A. Kumar et al. // arXiv, 2015. http://arxiv.org/abs/1506.07285.
11. Asynchronous Methods for Deep Reinforcement Learning / V. Mnih et al. // ArXiv, 2016. http://arxiv.org/abs/1602.01783.
12. Attention Is All You Need / A. Vaswani et al. // arXiv, 2017. http://arxiv.org/abs/1706.03762.
13. *Auer P., Cesa-Bianchi N., Fischer P.* Finite-Time Analysis of the Multiarmed Bandit Problem // Machine Learning, 2002, vol. 47, no. 2–3. — P. 235–256.
14. The Author-Topic Model for Authors and Documents / M. Rosen-Zvi et al. // Proc. 20th UAI, Arlington, VI, United States: AUAI Press, 2004. — P. 487–494.
15. *Ba J., Mnih V., Kavukcuoglu K.* Multiple Object Recognition with Visual Attention // arXiv, 2014. http://arxiv.org/abs/1412.7755.
16. *Ba L. J., Kiros R., Hinton G. E.* Layer Normalization // arXiv, 2016. http://arxiv.org/abs/1607.06450.
17. *Bachman P., Precup D.* Variational Generative Stochastic Networks with Collaborative Shaping // Proc. 32nd ICML, 2015. — P. 1964–1972.
18. Back-Propagation Applied to Handwritten Zip Code Recognition / Y. LeCun et al. // Neural Computation, 1989, vol. 1, no. 4. — P. 541–551.
19. A Backward Progression of Attentional Effects in the Ventral Stream / E. A. Buffalo et al. // Proc. National Academy of Sciences, 2010, vol. 107, no. 1. — P. 361–365.
20. *Badrinarayanan V., Kendall A., Cipolla R.* SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation // arXiv, 2015. http://arxiv.org/abs/1511.00561.
21. Bag of Tricks for Efficient Text Classification / A. Joulin et al. // arXiv, 2016. http://arxiv.org/abs/1607.01759.

22. *Bahdanau D., Cho K., Bengio Y.* Neural Machine Translation by Jointly Learning to Align and Translate // arXiv, 2014. http://arxiv.org/abs/1409.0473.
23. *Baillargeon R.* Infants' Physical World // Current Directions in Psychological Science, 2004, vol. 13, no. 3. — P. 89–94.
24. *Bain A.* The Senses and the Intellect, London: Parker, 1855.
25. *Baker C. L., Saxe R., Tenenbaum J. B.* Bayesian Theory of Mind: Modeling Joint Belief-Desire Attribution // Proc. 33th CogSci, 2011.
26. *Ballard D. H.* Modular Learning in Neural Networks // Proc. AAAI, 1987. — P. 279–284.
27. *Ballesteros M., Dyer C., Smith N. A.* Improved Transition-based Parsing by Modeling Characters instead of Words with LSTMs // Proc. EMNLP 2015, Lisbon, Portugal: ACL, 2015. — P. 349–359.
28. *Baltescu P., Blunsom P.* Pragmatic Neural Language Modelling in Machine Translation // NAACL HLT 2015, 2015. — P. 820–829.
29. *Banchs R. E.* Movie-DiC: A Movie Dialogue Corpus for Research and Development // Proc. 50th ACL: Short Papers - Volume 2, Stroudsburg, PA, USA: ACL, 2012. — P. 203–207.
30. *Banerjee S., Roy A.* Linear Algebra and Matrix Analysis for Statistics. Texts in Statistical Science, Chapman and Hall/CRC, 2014.
31. *Barber D., Bishop C.* Ensemble Learning in Bayesian Neural Networks // Neural Networks and Machine Learning, Springer, 1998. — P. 215–237.
32. *Baroni M., Zamparelli R.* Nouns Are Vectors, Adjectives Are Matrices: Representing Adjective-noun Constructions in Semantic Space // Proc. EMNLP 2010, Stroudsburg, PA, USA: ACL, 2010. — P. 1183–1193.
33. *Barto A. G., Sutton R. S., Anderson C. W.* Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems // Artificial Neural Networks / Piscataway, NJ, USA: IEEE Press, 1990. — P. 81–93.
34. Batch Normalized Recurrent Neural Networks / C. Laurent et al. // 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), March 2016. — P. 2657–2661.
35. *Baxter J., Bartlett P. L.* Infinite-Horizon Policy-Gradient Estimation // Journal of Artificial Intelligence Research, 2001, vol. 15, no. 1. — P. 319–350.
36. *Bayes T.* An Essay Towards Solving a Problem in the Doctrine of Chances // Philosophical Transactions of the Royal Society of London, 1763, vol. 53. — P. 370–418.
37. *Benenson R.* Are We There Yet? Crowdsourced List of State of the Art Results in Object Classification, 2016. http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html.
38. *Bengio Y.* Learning Deep Architectures for AI // Foundations and Trends in Machine Learning, 2009, vol. 2, no. 1. — P. 1–127.
39. *Bengio Y., Courville A. C., Vincent P.* Unsupervised Feature Learning and Deep Learning: A Review and New Perspectives // arXiv, 2012. http://arxiv.org/abs/1206.5538.
40. *Bengio Y., Ducharme R., Vincent P.* A Neural Probabilistic Language Model // Journal of Machine Learning Research, 2003, vol. 3. — P. 1137–1155.
41. *Bengio Y., Thibodeau-Laufer E., Yosinski J.* Deep Generative Stochastic Networks Trainable by Backprop // arXiv, 2013. http://arxiv.org/abs/1306.1091.
42. *Bian J., Gao B., Liu T.-Y.* Knowledge-Powered Deep Learning for Word Embedding // Machine Learning and Knowledge Discovery in Databases / Springer, 2014. — P. 132–148.
43. *Bickel S., Bruckner M., Scheffer T.* Discriminative Learning under Covariate Shift // Journal of Machine Learning Research, 2009, vol. 10, no. Sep. — P. 2137–2155.
44. *Bishop C. M.* Pattern Recognition and Machine Learning, Springer, 2006.
45. *Blei D. M., Jordan M. I., Paisley J. W.* Variational Bayesian Inference with Stochastic Search // Proc. 29th ICML, New York, NY, USA: ACM, 2012. — P. 1367–1374.

46. *Blei D. M., Lafferty J. D.* Correlated Topic Models // Advances in Neural Information Processing Systems, 2006, vol. 18.
47. *Blei D. M., Ng A. Y., Jordan M. I.* Latent Dirichlet allocation // Journal of Machine Learning Research, 2003, vol. 3, no. 4–5. — P. 993–1022.
48. BLEU: a Method for Automatic Evaluation of Machine Translation / K. Papineni et al. // Proc. 40th ACL, 2002. — P. 311–318.
49. *Bliss T. V., Collingridge G. L.* A Synaptic Model of Memory: Long-Term Potentiation in the Hippocampus // Nature, 1993, vol. 361, no. 6407. — P. 31–39.
50. *Bliss T. V., Lomo T.* Long-Lasting Potentiation of Synaptic Transmission in the Dentate Area of the Anaesthetized Rabbit Following Stimulation of the Perforant Path // Journal of Physiology, 1973, vol. 232, no. 2. — P. 331–356.
51. *Bloom P.* How Children Learn the Meanings of Words, Cambridge, MA: MIT Press, 2000.
52. *Bostrom N.* Superintelligence: Paths, Dangers, Strategies, Oxford, UK: Oxford University Press, 2014.
53. *Bostrom N., Yudkowsky E.* The Ethics of Artificial Intelligence // The Cambridge Handbook of Artificial Intelligence / New York: Cambridge University Press, 2014.
54. *Botha J. A., Blunsom P.* Compositional Morphology for Word Representations and Language Modelling // Proc. 31th ICML, 2014. — P. 1899–1907.
55. *Boulanger-lewandowski N., Bengio Y., Vincent P.* Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription // Proc. 29th ICML, New York, NY, USA: ACM, 2012. — P. 1159–1166.
56. *Boulanger-Lewandowski N., Bengio Y., Vincent P.* Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription // Proc. 29th ICML, icml.cc / Omnipress, 2012.
57. *Boureau Y.-L., Ponce J., LeCun Y.* A Theoretical Analysis of Feature Pooling in Visual Recognition. // Proc. 27th ICML, Omnipress, 2010. — P. 111–118.
58. *Bowman S. R., Potts C., Manning C. D.* Learning Distributed Word Representations for Natural Logic Reasoning // arXiv, 2014. http://arxiv.org/abs/1410.4176.
59. *Bowman S. R., Potts C., Manning C. D.* Recursive Neural Networks for Learning Logical Semantics // arXiv, 2014. http://arxiv.org/abs/1406.1827.
60. *Bradshaw R., Citro C., Seljebotn D.* Cython: The Best of Both Worlds // CiSE 2011 Special Python Issue, 2010. — P. 25.
61. Breaking Sticks and Ambiguities with Adaptive Skip-gram / S. Bartunov et al. // Proc. 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016, 2016. — P. 130–138.
62. *Bride A., Van de Cruys T., Asher N.* A Generalisation of Lexical Functions for Composition in Distributional Semantics // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 281–291.
63. *Brown N., Sandholm T.* Safe and Nested Endgame Solving for Imperfect-Information Games // Proc. AAAI-17 Workshop on Computer Poker and Imperfect Information Games, 2017.
64. *Brownlee J.* Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras, 2016. http://machinelearningmastery.com/.
65. *Buchanan B. G., Shortliffe E. H.* Rule Based Expert Systems: The Mycin Experiments of the Stanford Heuristic Programming Project (The Addison-Wesley Series in Artificial Intelligence), Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984.
66. Building Machines That Learn and Think Like People / B. M. Lake et al. // Behavioral and Brain Sciences, Nov 2016. — P. 1–101.
67. *Buys J., Blunsom P.* Generative Incremental Dependency Parsing with Neural Networks // Proc. 53rd ACL and the 7th International Joint Conference on Natural Language Processing

of the Asian Federation of Natural Language Processing, Vol. 2: Short Papers, 2015. — P. 863–869.
68. *Calafiore G. C., El Ghaoui L.* Optimization Models, Cambridge University Press, 2014.
69. CAN: Creative Adversarial Networks, Generating "Art" by Learning About Styles and Deviating from Style Norms / A. M. Elgammal et al. // arXiv, 2017. http://arxiv.org/abs/1706.07068.
70. *Caporale N., Dan Y.* Spike Timing–Dependent Plasticity: A Hebbian Learning Rule // Annual Review of Neuroscience, 2008, vol. 31. — P. 25–46.
71. *Carandini M.* Area V1 // Scholarpedia / 2012. — P. 12105. http://www.scholarpedia.org/article/Area_V1.
72. *Carreira-Perpinan M. A., Hinton G.* On Contrastive Divergence Learning // AISTATS / vol. 10, 2005. — P. 33–40.
73. *Chang J., Blei D. M.* Hierarchical Relational Models for Document Networks // Annals of Applied Statistics, 2010, vol. 4, no. 1. — P. 124–150.
74. Character-Aware Neural Language Models / Y. Kim et al. // arXiv, 2015. http://arxiv.org/abs/1508.06615.
75. *Chen J., Deng L.* A New Method for Learning Deep Recurrent Neural Networks // arXiv, 2013. http://arxiv.org/abs/1311.6091.
76. *Chen S. F., Goodman J.* An Empirical Study of Smoothing Techniques for Language Modeling // Proc. 34th ACL, Stroudsburg, PA, USA: ACL, 1996. — P. 310–318.
77. *Chintala S., LeCun Y.* A Path to Unsupervised Learning through Adversarial Networks, 2016. https://code.facebook.com/posts/1587249151575490.
78. *Cho K.* Introduction to Neural Machine Translation with GPUs, 2015. https://devblogs.nvidia.com/parallelforall/introduction-neural-machine-translation-with-gpus/.
79. *Chollet F.* Keras, https://github.com/fchollet/keras, 2015.
80. *Chomsky N.* Syntactic Structures, Mouton & Co., 1957.
81. *Chung J., Cho K., Bengio Y.* A Character-level Decoder without Explicit Segmentation for Neural Machine Translation // arXiv, 2016. http://arxiv.org/abs/1603.06147.
82. *Churchland P. S., Ramachandran V. S., Sejnowski T. J.* A Critique of Pure Vision // Large-Scale Neuronal Theories of the Brain / MIT Press, 1993. — P. 23.
83. *Clark S., Coecke B., Sadrzadeh M.* A Compositional Distributional Model of Meaning // Proc. Second Symposium on Quantum Interaction (QI-2008), 2008. — P. 133–140.
84. *Clark S., Coecke B., Sadrzadeh M.* Mathematical Foundations for a Compositional Distributed Model of Meaning // Linguistic Analysis, 2011, vol. 36, no. 1-4. — P. 345–384.
85. Class-based N-gram Models of Natural Language / P. F. Brown et al. // Comput. Linguist., 1992, vol. 18, no. 4. — P. 467–479.
86. *Clevert D., Unterthiner T., Hochreiter S.* Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs) // arXiv, 2015. http://arxiv.org/abs/1511.07289.
87. *Coecke B., Sadrzadeh M., Clark S.* Mathematical Foundations for a Compositional Distributional Model of Meaning // arXiv, 2010. http://arxiv.org/abs/1003.4394.
88. *Collobert R., Bengio S.* Links Between Perceptrons, MLPs and SVMs // Proc. 21st ICML, New York, NY, USA: ACM, 2004. — P. 23–.
89. *Collobert R., Bengio S., Marithoz J.* Torch: A Modular Machine Learning Software Library, 2002.
90. *Collobert R., Kavukcuoglu K., Farabet C.* Torch7: A Matlab-like Environment for Machine Learning // BigLearn, NIPS Workshop, 2011.
91. Combining Model-Based and Model-Free Updates for Trajectory-Centric Reinforcement Learning / Y. Chebotar et al. // arXiv, 2017. http://arxiv.org/abs/1703.03078.
92. Concrete Sentence Spaces for Compositional Distributional Models of Meaning / E. Grefenstette et al. // Proc. 9th International Conference on Computational Semantics (IWCS11), 2011. — P. 125–134.

93. Concrete Sentence Spaces for Compositional Distributional Models of Meaning / E. Grefenstette et al. // Computing Meaning / Springer, 2014. — P. 71–86.
94. Conditional Image Generation with PixelCNN Decoders / A. van den Oord et al. // arXiv, 2016. http://arxiv.org/abs/1606.05328.
95. A Context-Aware Topic Model for Statistical Machine Translation / J. Su et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 229–238.
96. Context-Dependent Pre-Trained Deep Neural Networks for Large Vocabulary Speech Recognition / G. Dahl et al. / vol. 20, 2012. — P. 30–42.
97. The Cornucopia of Meaningful Leads: Applying Deep Adversarial Autoencoders for New Molecule Development in Oncology / A. Kadurin et al. // Oncotarget, 2017, vol. 8. — P. 10883–10890.
98. Covariate Shift by Kernel Mean Matching / A. Gretton et al. // Dataset Shift in Machine Learning, 2009, vol. 3, no. 4. — P. 5.
99. *Cox R. T.* The Algebra of Probable Inference, Johns Hopkins Press Baltimore, 1961. — 114 P.
100. *Creswell A., Bharath A. A.* Denoising Adversarial Autoencoders // arXiv, 2017. http://arxiv.org/abs/1703.01220.
101. CS231n Convolutional Neural Networks for Visual Recognition. http://cs231n.github.io/neural-networks-3/#second.
102. cuDNN: Efficient Primitives for Deep Learning / S. Chetlur et al. // arXiv, 2014. http://arxiv.org/abs/1410.0759.
103. Dataset Shift in Machine Learning / J. Quionero-Candela et al., The MIT Press, 2009.
104. *Daugman J. G.* Uncertainty Relation for Resolution in Space, Spatial Frequency, and Orientation Optimized by Two-Dimensional Visual Cortical Filters // Journal of the Optical Society of America A, 1985, vol. 2, no. 7. — P. 1160–1169.
105. *Dayan P., Abott L.* Theoretical Neuroscience, Cambridge, MA, USA: MIT Press, 2001.
106. Deep Generative Adversarial Networks for Compressed Sensing Automates MRI / M. Mardani et al. // arXiv, 2017. http://arxiv.org/abs/1706.00051.
107. Deep Learning for Visual Understanding: A Review / Y. Guo et al. // Neurocomputing, 2016, vol. 187. — P. 27 – 48, Recent Developments on Deep Big Vision.
108. *Bergstra J., Breuleux O. et al.* Deep Learning Tutorial: Theano Documentation, 2015.
109. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups / G. Hinton et al. // IEEE Signal Processing Magazine, Nov 2012, vol. 29, no. 6. — P. 82–97.
110. Deep Reinforcement Learning for Robotic Manipulation / S. Gu et al. // arXiv, 2016. http://arxiv.org/abs/1610.00633.
111. Deep Residual Learning for Image Recognition / K. He et al. // Proc. 2016 CVPR, 2016. — P. 770–778.
112. DeepFace: Closing the Gap to Human-Level Performance in Face Verification / Y. Taigman et al. // Proc. 2014 IEEE Conference on Computer Vision and Pattern Recognition, Washington, DC, USA: IEEE Computer Society, 2014. — P. 1701–1708.
113. DeepStack: Expert-Level Artificial Intelligence in Heads-Up No-Limit Poker / M. Moravcik et al. // Science, 2017, vol. 356, no. 6337. — P. 508–513.
114. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification / K. He et al. // Proc. ICCV 2015, 2015. — P. 1026–1034.
115. *Deng L.* A Tutorial Survey of Architectures, Algorithms, and Applications for Deep Learning // APSIPA Transactions on Signal and Information Processing, 2014.
116. *Deng L., Hinton G., Kingsbury B.* New Types of Deep Neural Network Learning for Speech Recognition and Related Applications: An Overview // IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), May 2013, 2013.

117. *Deng L., Yu D.* Deep Learning: Methods and Applications // Foundations and Trends in Signal Processing, 2014, vol. 7, no. 3–4. — P. 197–387.
118. Dependency-based Convolutional Neural Networks for Sentence Embedding / M. Ma et al. // Proc. ACL 2015, Vol. 2: Short Papers, 2015. — P. 174.
119. Developments and Directions in Speech Recognition and Understanding, Part 1 [DSP Education] / J. M. Baker et al. // IEEE Signal Processing Magazine, May 2009, vol. 26, no. 3. — P. 75–80.
120. Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics / J. Mahler et al. // arXiv, 2017. http://arxiv.org/abs/1703.09312.
121. *Dinh L., Sohl-Dickstein J., Bengio S.* Density Estimation using Real NVP // arXiv, 2016. http://arxiv.org/abs/1605.08803.
122. Direct Importance Estimation with Model Selection and its Application to Covariate Shift Adaptation / M. Sugiyama et al. // Advances in neural information processing systems, 2008. — P. 1433–1440.
123. Distributed Representations of Words and Phrases and their Compositionality / T. Mikolov et al. // arXiv, 2013. http://arxiv.org/abs/1310.4546.
124. *Doersch C.* Tutorial on Variational Autoencoders // ArXiv e-prints, 2016.
125. *Domingos P.* The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World. Basic Books, Basic Books, 2015.
126. *Downey A. B.* Think Python — How to Think Like a Computer Scientist, O'Reilly, 2012.
127. DRAW: A Recurrent Neural Network For Image Generation / K. Gregor et al. // arXiv, 2015. http://arxiv.org/abs/1502.04623.
128. *Dreyfus S. E.* The Computational Solution of Optimal Control Problems with Time Lag // IEEE Transactions on Automatic Control, 1973, vol. 18(4). — P. 383–385.
129. Dropout: A Simple Way to Prevent Neural Networks from Overfitting / N. Srivastava et al. // Journal of Machine Learning Research, 2014, vol. 15, no. 1. — P. 1929–1958.
130. druGAN: An Advanced Generative Adversarial Autoencoder Model for de Novo Generation of New Molecules with Desired Molecular Properties in Silico / A. Kadurin et al. // Molecular Pharmaceutics, 2017.
131. *Duchi J., Hazan E., Singer Y.* Adaptive subgradient methods for online learning and stochastic optimization // Journal of Machine Learning Research, 2011, vol. 12, no. Jul. — P. 2121–2159.
132. *Duffy S. A., Henderson J. M., Morris R. K.* Semantic facilitation of lexical access during sentence processing // Journal of Experimental Psychology: Learning, Memory, and Cognition, 1989, vol. 15. — P. 791–801.
133. *Durrett G., Klein D.* Neural CRF parsing // arXiv, 2015. http://arxiv.org/abs/1507.03641.
134. Dynamic Pooling and Unfolding Recursive Autoencoders for Paraphrase Detection / R. Socher et al. // Advances in Neural Information Processing Systems, 2011. — P. 801–809.
135. Efficient BackProp / Y. LeCun et al. // Neural Networks: Tricks of the Trade, London, UK, UK: Springer-Verlag, 1998. — P. 9–50.
136. Efficient Backprop / Y. LeCun et al. // Neural Networks: Tricks of the Trade / Springer Berlin Heidelberg, 2012. — P. 9–48.
137. Efficient Estimation of Word Representations in Vector Space / T. Mikolov et al. // arXiv, 2013. http://arxiv.org/abs/1301.3781.
138. *El Hihi S., Bengio Y.* Hierarchical Recurrent Neural Networks for Long-Term Dependencies, 1996.
139. *Elman J. L.* Finding Structure in Time // Cognitive Science, 1990, vol. 14, no. 2. — P. 179–211.
140. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling / J. Chung et al. // arXiv, 2014. http://arxiv.org/abs/1412.3555.

141. Encoding Source Language with Convolutional Neural Network for Machine Translation / F. Meng et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 20–30.
142. End to End Learning for Self-Driving Cars / M. Bojarski et al. // arXiv, 2016. http://arxiv.org/abs/1604.07316.
143. End-to-end Training of Deep Visuomotor Policies / S. Levine et al. // Journal of Machine Learning Research, 2016, vol. 17, no. 1. — P. 1334–1373.
144. Enriching Word Vectors with Subword Information / P. Bojanowski et al. // arXiv, 2016. http://arxiv.org/abs/1607.04606.
145. Euclidean Embedding of Co-occurrence Data / A. Globerson et al. // Journal of Machine Learning Research, 2007, vol. 8. — P. 2265–2295.
146. Exploring the Limits of Language Modeling / R. J zefowicz et al. // arXiv, 2016. http://arxiv.org/abs/1602.02410.
147. Extensions of Recurrent Neural Network Language Model / T. Mikolov et al. // Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, 2011. — P. 5528–5531.
148. Extracting and Composing Robust Features with Denoising Autoencoders / P. Vincent et al. // Proc. 25th ICML, New York, NY, USA: ACM, 2008. — P. 1096–1103.
149. Extractive Summarization using Continuous Vector Space Models / M. Kageback et al. // Proc. 2nd Workshop on Continuous Vector Space Models and their Compositionality (CVSC)@ EACL, 2014. — P. 31–39.
150. *Farlow S. J.* Self-Organizing Methods in Modeling: Gmdh Type Algorithms, New York, NY, USA: Marcel Dekker, Inc., 1984.
151. Fast and Robust Neural Network Joint Models for Statistical Machine Translation / J. Devlin et al. // Proc. 52nd ACL, Vol. 1: Long Papers, Baltimore, Maryland: ACL, June 2014. — P. 1370–1380.
152. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks / S. Ren et al. // arXiv, 2015. http://arxiv.org/abs/1506.01497.
153. FastText.zip: Compressing Text Classification Models / A. Joulin et al. // arXiv, 2016. http://arxiv.org/abs/1612.03651.
154. *Fellbaum C.* WordNet and wordnets // Encyclopedia of Language and Linguistics, Oxford: Elsevier, 2005. — P. 665–670.
155. *Felzenszwalb P. F., Girshick R. B., McAllester D. A.* Cascade Object Detection with Deformable Part Models // CVPR, IEEE Computer Society, 2010. — P. 2241–2248.
156. Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation / W. Ling et al. // Proc. EMNLP 2015, Lisbon, Portugal: ACL, 2015. — P. 1520–1530.
157. Findings of the 2016 Conference on Machine Translation / O. Bojar et al. // Proc. First Conference on Machine Translation, Berlin, Germany: ACL, August 2016. — P. 131–198.
158. *Finn C., Levine S., Abbeel P.* Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization // Proc. 33nd ICML, 2016. — P. 49–58.
159. *Finnson H.* Generalized Monte-Carlo Tree Search Extensions for General Game Playing // Proc. 26th AAAI Conference on Artificial Intelligence, 2012. — P. 1550–1556.
160. *Finnson H.* Simulation-Based General Game Playing: Ph.D. thesis / Reykjavik University, 2012.
161. *Firat O., Cho K., Bengio Y.* Multi-Way, Multilingual Neural Machine Translation with A Shared Attention Mechanism // arXiv, 2016. http://arxiv.org/abs/1601.01073.
162. Flexible, High Performance Convolutional Neural Networks for Image Classification / D. C. Ciresan et al. // Proc. 22nd International Joint Conference on Artificial Intelligence – Volume Two, AAAI Press, 2011. — P. 1237–1242.

163. Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge / K. Bollacker et al. // Proc. 2008 ACM SIGMOD International Conference on Management of Data, New York, NY, USA: ACM, 2008. — P. 1247–1250.
164. *Frey B.* Graphical Models for Machine Learning and Digital Communication, Cambridge, MA: MIT Press, 1998.
165. *Fried D., Polajnar T., Clark S.* Low-Rank Tensors for Verbs in Compositional Distributional Semantics // Proc. 53rd ACL and the 7th IJCNLP, Vol. 2: Short Papers, Beijing, China: ACL, 2015. — P. 731–736.
166. *Fukushima K.* Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position — Neocognitron // Transactions of the IECE, 1979, vol. J62-A(10). — P. 658–665.
167. *Fukushima K.* Neocognitron: A Self-Organizing Neural Network for a Mechanism of Pattern Recognition Unaffected by Shift in Position // Biological Cybernetics, 1980, vol. 36, no. 4. — P. 193–202.
168. Gabor Analysis and Algorithms / Ed. by H. G. Feichtinger, T. Strohmer. Applied and Numerical Harmonic Analysis, Birkhauser, 1998.
169. *Gal Y.* A Theoretically Grounded Application of Dropout in Recurrent Neural Networks // arXiv:1512.05287, 2015.
170. *Gal Y.* Uncertainty in Deep Learning: Ph.D. thesis / University of Cambridge, 2016.
171. *Gal Y., Ghahramani Z.* Dropout as a Bayesian Approximation: Insights and Applications // Deep Learning Workshop, ICML, 2015.
172. *Gal Y., Ghahramani Z.* On Modern Deep Learning and Variational Inference // Advances in Approximate Bayesian Inference workshop, NIPS, 2015.
173. *Ganitkevitch J., Durme B. V., Callison-Burch C.* PPDB: The Paraphrase Database // Proc. HLT-NAACL, ACL, 2013. — P. 758–764.
174. *Gefter A.* The Man Who Tried to Redeem the World with Logic // Nautilus, 2015, vol. 21. — P. 106–154.
175. *Geitgey A.* Abusing Generative Adversarial Networks to Make 8-bit Pixel Art, 2017. https://medium.com/@ageitgey/.
176. Generative Adversarial Networks / I. J. Goodfellow et al. // ArXiv e-prints, 2014.
177. *Gers F. A., Schmidhuber J.* Recurrent Nets that Time and Count // Neural Networks, 2000. IJCNN 2000, Proc. IEEE-INNS-ENNS International Joint Conference on / vol. 3, 2000. — P. 189–194.
178. *Gers F. A., Schmidhuber J., Cummins F.* Learning to Forget: Continual Prediction with LSTM // Neural Computation, 2000, vol. 12, no. 10. — P. 2451–2471.
179. *Glorot X., Bengio Y.* Understanding the Difficulty of Training Deep Feedforward Neural Networks // International conference on artificial intelligence and statistics, 2010. — P. 249–256.
180. *Glorot X., Bordes A., Bengio Y.* Deep Sparse Rectifier Networks // AISTATS / vol. 15, 2011. — P. 315–323.
181. Going Deeper with Convolutions / C. Szegedy et al. // arXiv, 2014, vol. abs/1409.4842. http://arxiv.org/abs/1409.4842.
182. *Goldberg Y.* A Primer on Neural Network Models for Natural Language Processing // arXiv, 2015. http://arxiv.org/abs/1510.00726.
183. *Goldberg Y., Levy O.* Word2vec Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method // arXiv, 2014. http://arxiv.org/abs/1402.3722.
184. *Goodfellow I., Bengio Y., Courville A.* Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.
185. *Goodfellow I. J.* NIPS 2016 Tutorial: Generative Adversarial Networks // arXiv, 2017. http://arxiv.org/abs/1701.00160.

186. *Goodman J. T.* A Bit of Progress in Language Modeling // Comput. Speech Lang., 2001, vol. 15, no. 4. — P. 403–434.
187. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation / Y. Wu et al. // arXiv, 2016. http://arxiv.org/abs/1609.08144.
188. Gradient-Based Learning Applied to Document Recognition / Y. LeCun et al. // Proc. IEEE, 1998, vol. 86, no. 11. — P. 2278–2324.
189. *Granberg D.* The Monty Hall Dilemma: A Cognitive Illusion Par Excellence, Lumad/CreateSpace, 2014.
190. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions / S. Gelly et al. // Communications of the ACM, 2012, vol. 55, no. 3. — P. 106–113.
191. *Graves A.* Practical Variational Inference for Neural Networks // Advances in Neural Information Processing Systems 24 / Curran Associates, Inc., 2011. — P. 2348–2356.
192. *Graves A.* Generating Sequences With Recurrent Neural Networks // arXiv, 2013. http://arxiv.org/abs/1308.0850.
193. *Graves A., Schmidhuber J.* Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures // Neural Networks, 2005, vol. 18, no. 5-6. — P. 602–610.
194. *Graves A., Wayne G., Danihelka I.* Neural Turing Machines // arXiv, 2014. http://arxiv.org/abs/1410.5401.
195. Greedy Layer-Wise Training of Deep Networks / Y. Bengio et al. // Proc. 19th NIPS, Cambridge, MA, USA: MIT Press, 2006. — P. 153–160.
196. *Grefenstette E.* Towards a Formal Distributional Semantics: Simulating Logical Calculi with Tensors // arXiv, 2013. http://arxiv.org/abs/1304.5823.
197. *Grefenstette E., Sadrzadeh M.* Experimental Support for a Categorical Compositional Distributional Model of Meaning // Proc. EMNLP 2011, Stroudsburg, PA, USA: ACL, 2011. — P. 1394–1404.
198. *Griffiths T., Steyvers M.* Finding Scientific Topics // Proc. National Academy of Sciences, 2004, vol. 101 (Suppl. 1). — P. 5228–5335.
199. GSNs : Generative Stochastic Networks / G. Alain et al. // arXiv, 2015. http://arxiv.org/abs/1503.05571.
200. *Guenin B., Konemann J., Tun el L.* A Gentle Introduction to Optimization, Cambridge University Press, 2014.
201. *Guo H.* Generating Text with Deep Reinforcement Learning // arXiv, 2015. http://arxiv.org/abs/1510.09292.
202. *Hall D., Durrett G., Klein D.* Less Grammar, More Features // Proc. 52nd ACL, Vol. 1: Long Papers, Baltimore, Maryland: ACL, June 2014. — P. 228–237.
203. *Hamlin J. K.* Moral Judgment and Action in Preverbal Infants and Toddlers // Current Directions in Psychological Science, 2013, vol. 22, no. 3. — P. 186–193.
204. *Han A. L. F., Wong D. F., Chao L. S.* LEPOR: A Robust Evaluation Metric for Machine Translation with Augmented Factors // Proceedings of COLING 2012: Posters, Mumbai, India: The COLING 2012 Organizing Committee, December 2012. — P. 441–450.
205. Handwritten Digit Recognition with a Back-Propagation Network / Y. LeCun et al. // Advances in Neural Information Processing Systems 2, Morgan Kaufmann, 1990. — P. 396–404.
206. *Hanson R., Yudkowsky E.* The Hanson-Yudkowsky AI-Foom Debate, Berkeley, CA: Machine Intelligence Research Institute, 2013.
207. *Hanson S. J., Pratt L. Y.* Comparing Biases for Minimal Network Construction with Back-Propagation // Advances in Neural Information Processing Systems (NIPS) 1, San Mateo, CA: Morgan Kaufmann, 1989. — P. 177–185.
208. *Hastad J.* Computational Limitations of Small-Depth Circuits, Cambridge, MA, USA: MIT Press, 1987.

209. *Hebb D. O.* The Organization of Behavior, Wiley, New York, 1949.
210. *Heess N., Silver D., Teh Y. W.* Actor-Critic Reinforcement Learning with Energy-Based Policies. // EWRL / vol. 24 of *JMLR Proceedings*, JMLR.org, 2012. — P. 43–58.
211. *Hermann K. M., Blunsom P.* Multilingual Models for Compositional Distributed Semantics // Proc. 52nd ACL, Vol. 1: Long Papers, Baltimore, Maryland: ACL, 2014. — P. 58–68.
212. A Hierarchical Latent Variable Encoder-Decoder Model for Generating Dialogues / I. V. Serban et al. // Proc. 31st AAAI, 2017. — P. 3295–3301.
213. Hierarchical Neural Language Models for Joint Representation of Streaming Documents and Their Content / N. Djuric et al. // Proc. 24th WWW, New York, NY, USA: ACM, 2015. — P. 248–255.
214. Hierarchical Neural Network Generative Models for Movie Dialogues / I. V. Serban et al. // arXiv, 2015. http://arxiv.org/abs/1507.04808.
215. A Hierarchical Recurrent Encoder-Decoder for Generative Context-Aware Query Suggestion / A. Sordoni et al. // Proc. 24th ACM International on Conference on Information and Knowledge Management, New York, NY, USA: ACM, 2015. — P. 553–562.
216. High-Performance Neural Networks for Visual Object Classification / D. C. Ciresan et al. // arXiv, 2011. http://arxiv.org/abs/1102.0183.
217. *Hindupur A.* The GAN Zoo: A List of All Named GANs, 2017. https://github.com/hindupuravinash/the-gan-zoo.
218. *Hinton G.* How to Do Backpropagation in a Brain, Invited talk at the NIPS'2007 Deep Learning Workshop, 2007.
219. *Hinton G. E.* Training Products of Experts by Minimizing Contrastive Divergence // Neural Computation, 2002, vol. 14, no. 8. — P. 1771–1800.
220. *Hinton G. E.* Learning Multiple Layers of Representation // Trends in Cognitive Sciences, 2007, vol. 11. — P. 428–434.
221. *Hinton G. E.* A Practical Guide to Training Restricted Boltzmann Machines. // Neural Networks: Tricks of the Trade (2nd ed.) / Springer, 2012. — P. 599–619.
222. *Hinton G. E., McClelland J. L.* Learning Representations by Recirculation // Neural Information Processing Systems / American Institute of Physics, 1988. — P. 358–366.
223. *Hinton G. E., Osindero S., Teh Y.-W.* A Fast Learning Algorithm for Deep Belief Nets // Neural Computation, 2006, vol. 18, no. 7. — P. 1527–1554.
224. *Hinton G. E., Salakhutdinov R. R.* Reducing the Dimensionality of Data with Neural Networks // Science, 2006, vol. 313, no. 5786. — P. 504–507.
225. *Hinton G. E., Sejnowski T. J.* Learning and Relearning in Boltzmann Machines // Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1 / Cambridge, MA, USA: MIT Press, 1986. — P. 282–317.
226. *Hinton G. E., van Camp D.* Keeping the Neural Networks Simple by Minimizing the Description Length of the Weights // Proc. 6th CoLT, New York, NY, USA: ACM, 1993. — P. 5–13.
227. *Hochreiter S., Schmidhuber J.* Long Short-Term Memory: Tech. Rep. FKI-207-95: Fakultat fur Informatik, Technische Universitat Munchen, 1995.
228. *Hochreiter S., Schmidhuber J.* Long Short-Term Memory // Neural Computation, 1997, vol. 9, no. 8. — P. 1735–1780.
229. *Hoffmann T.* Unsupervised Learning by Probabilistic Latent Semantic Analysis // Machine Learning, 2001, vol. 42, no. 1. — P. 177–196.
230. *Hopfield J. J.* Neural Networks and Physical Systems with Emergent Collective Computational Abilities // Proc. of the National Academy of Sciences, 1982, vol. 79. — P. 2554–2558.
231. *Hornik K., Stinchcombe M., White H.* Multilayer Feedforward Networks are Universal Approximators // Neural Networks, 1989, vol. 2, no. 5. — P. 359–366.

232. How NOT To Evaluate Your Dialogue System: An Empirical Study of Unsupervised Evaluation Metrics for Dialogue Response Generation / C. Liu et al. // Proc. EMNLP 2016, 2016. — P. 2122–2132.
233. *Howard R. A.* Dynamic Programming and Markov Processes, Cambridge, MA: MIT Press, 1960.
234. *Hubel D. H.* Eye, Brain, and Vision, 2nd edition, W. H. Freeman, 1995.
235. *Hubel D. H., Wiesel T.* Receptive Fields, Binocular Interaction, and Functional Architecture in the Cat's Visual Cortex // Journal of Physiology (London), 1962, vol. 160. — P. 106–154.
236. *Hubel D. H., Wiesel T. N.* Receptive Fields and Functional Architecture of Monkey Striate Cortex // Journal of Physiology, 1968, vol. 195, no. 1. — P. 215–243.
237. *Hubel D. H., Wiesel T. N.* Brain and Visual Perception: The Story of a 25-Year Collaboration, Oxford University Press, 2004.
238. Human-Level Control through Deep Reinforcement Learning / V. Mnih et al. // Nature, 2015, vol. 518, no. 7540. — P. 529–533.
239. Humans Predict Liquid Dynamics using Probabilistic Simulation / C. Bates et al. // Proc. CogSci 2015, 2015.
240. Hybrid Computing Using a Neural Network with Dynamic External Memory / A. Graves et al. // Nature, 2016, vol. 538, no. 7626. — P. 471–476.
241. *Hyvarinen A., Karhunen J., Oja E.* Independent Component Analysis, New York: Wiley, 2001.
242. Identity Mappings in Deep Residual Networks / K. He et al. // arXiv, 2016. http://arxiv.org/abs/1603.05027.
243. Image Processing of Human Corneal Endothelium Based on a Learning Network / W. Zhang et al. // Applied Optics, 1991, vol. 30, no. 29.
244. ImageNet Large Scale Visual Recognition Challenge / O. Russakovsky et al. // International Journal of Computer Vision (IJCV), 2015, vol. 115, no. 3. — P. 211–252.
245. Improved Techniques for Training GANs / T. Salimans et al. // arXiv, 2016. http://arxiv.org/abs/1606.03498.
246. Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors / G. E. Hinton et al. // arXiv, 2012. http://arxiv.org/abs/1207.0580.
247. Indexing by Latent Semantic Analysis / S. Deerwester et al. // Journal of the American Society for Information Science, 1990, vol. 41, no. 6. — P. 391–407.
248. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets / X. Chen et al. // arXiv, 2016. http://arxiv.org/abs/1606.03657.
249. Intriguing Properties of Neural Networks / C. Szegedy et al. // arXiv, 2013. http://arxiv.org/abs/1312.6199.
250. An Introduction to MCMC for Machine Learning / C. Andrieu et al. // Machine Learning, Jan 2003, vol. 50, no. 1. — P. 5–43.
251. Invertible Conditional GANs for Image Editing / G. Perarnau et al. // arXiv, 2016. http://arxiv.org/abs/1611.06355.
252. *Ioffe S., Szegedy C.* Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift // Proc. 32nd ICML, 2015. — P. 448–456.
253. *Irsoy O., Cardie C.* Opinion Mining with Deep Recurrent Neural Networks // Proc. EMNLP, 2014. — P. 720–728.
254. *Itti L., Koch C., Niebur E.* A Model of Saliency-Based Visual Attention for Rapid Scene Analysis // IEEE Trans. Pattern Anal. Machine Intelling, 1998, vol. 20, no. 11. — P. 1254–1259.
255. *Ivakhnenko A. G.* The Group Method of Data Handling – a Rival of the Method of Stochastic Approximation // Soviet Automatic Control, 1968, vol. 13, no. 3. — P. 43–55.
256. *Ivakhnenko A. G.* Polynomial Theory of Complex Systems // IEEE Transactions on Systems, Man and Cybernetics, 1971, no. 4. — P. 364–378.

257. *Ivakhnenko A. G., Lapa V. G., McDonough R. N.* Cybernetics and Forecasting Techniques, American Elsevier, NY, 1967.
258. *Jaeger H.* Discovering Multiscale Dynamical Features with Hierarchical Echo State Networks: Technical Report 10, Bremen, Germany: School of Engineering and Science, Jacobs University, 2007.
259. *Jain V., Seung S.* Natural Image Denoising with Convolutional Networks // Advances in Neural Information Processing Systems (NIPS) 21 / Curran Associates, Inc., 2009. — P. 769–776.
260. *Jaitly N., Hinton G. E.* Learning a Better Representation of Speech Soundwaves Using Restricted Boltzmann Machines // ICASSP, IEEE, 2011. — P. 5884–5887.
261. *Jeffreys H.* Theory of Probability, Oxford: Oxford University Press, 1939.
262. Joint Learning of Character and Word Embeddings / X. Chen et al. // Proc. 24th International Conference on Artificial Intelligence, AAAI Press, 2015. — P. 1236–1242.
263. *Jonas E., Kording K.* Could a Neuroscientist Understand a Microprocessor? // bioRxiv, 2016.
264. *Jordan M. I.* Serial Order: A Parallel Distributed Processing Approach: Tech. Rep. ICS Report 8604: Institute for Cognitive Science, University of California, San Diego, 1986.
265. *Jordan M. I.* Attractor Dynamics and Parallelism in a Connectionist Sequential Machine // Artificial Neural Networks / Piscataway, NJ, USA: IEEE Press, 1990. — P. 112–127.
266. *Jordan M. I.* Serial order: A parallel distributed processing approach // Advances in Psychology, 1997, vol. 121. — P. 471–495.
267. *J zefowicz R., Zaremba W., Sutskever I.* An Empirical Exploration of Recurrent Network Architectures // Proc. 32nd ICML, 2015. — P. 2342–2350.
268. *Kaku M.* Интервью, http://www.wnyc.org/story/michio-kaku-explores-human-brain/, 2014.
269. *Kalchbrenner N., Blunsom P.* Recurrent Continuous Translation Models // EMNLP / vol. 3, 2013. — P. 413.
270. *Kalchbrenner N., Blunsom P.* Recurrent Convolutional Neural Networks for Discourse Compositionality // arXiv, 2013. http://arxiv.org/abs/1306.3584.
271. *Kalchbrenner N., Grefenstette E., Blunsom P.* A Convolutional Neural Network for Modelling Sentences // arXiv, 2014. http://arxiv.org/abs/1404.2188.
272. *Karpathy A.* What I Learned from Competing Against a ConvNet on ImageNet, 2014. http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/.
273. *Karpathy A.* The Unreasonable Effectiveness of Recurrent Neural Networks, 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/.
274. *Karpathy A., Fei-Fei L.* Deep Visual-Semantic Alignments for Generating Image Descriptions // Proc. IEEE Conference on Computer Vision and Pattern Recognition, 2015. — P. 3128–3137.
275. *Kartsaklis D., Sadrzadeh M., Pulman S.* A Unified Sentence Space for Categorical Distributional-Compositional Semantics: Theory and Experiments // Proceedings of 24th International Conference on Computational Linguistics (COLING): Posters, Mumbai, India: 2012. — P. 549–558.
276. *Kendall A., Badrinarayanan V., Cipolla R.* Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding // arXiv, 2015. http://arxiv.org/abs/1511.02680.
277. *Kendall A., Cipolla R.* Modelling Uncertainty in Deep Learning for Camera Relocalization // arXiv, 2015. http://arxiv.org/abs/1509.05909.
278. *Kingma D., Ba J.* Adam: A Method for Stochastic Optimization // arXiv, 2014. http://arxiv.org/abs/1412.6980.

279. *Kingma D. P., Salimans T., Welling M.* Variational Dropout and the Local Reparameterization Trick // Advances in Neural Information Processing Systems 28 / Curran Associates, Inc., 2015. — P. 2575–2583.
280. *Kingma D. P., Welling M.* Auto-Encoding Variational Bayes // ArXiv e-prints, 2013.
281. *Kneser R., Ney H.* Improved Backing-Off for M-Gram Language Modeling // Proc. ICASSP-95 / vol. 1, 1995. — P. 181–184.
282. *Knudsen E. I.* Fundamental Components of Attention // Annual Review of Neuroscience, 2007, vol. 30, no. 1. — P. 57–78.
283. *Ko J., Fox D.* GP-BayesFilters: Bayesian Filtering using Gaussian Process Prediction and Observation Models // Proc. IROS 2008, 2008. — P. 3471–3476.
284. *Krizhevsky A., Sutskever I., Hinton G. E.* ImageNet Classification with Deep Convolutional Neural Networks // Advances in Neural Information Processing Systems 25 / Curran Associates, Inc., 2012. — P. 1097–1105.
285. *Krogh A., Hertz J. A.* A Simple Weight Decay can Improve Generalization // Advances in Neural Information Processing Systems 4, Morgan Kaufmann, 1992. — P. 950–957.
286. *Kullback S.* Information Theory and Statistics, John Wiley & Sons, 1959.
287. *Kullback S., Leibler R. A.* On Information and Sufficiency // Ann. Math. Statist., 1951, vol. 22, no. 1. — P. 79–86.
288. *Lacoste-Julien S., Sha F., Jordan M. I.* DiscLDA: Discriminative Learning for Dimensionality Reduction and Classification // Advances in Neural Information Processing Systems, 2008, vol. 20.
289. *Lafferty J., McCallum A., Pereira F. C.* Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data, 2001.
290. *Lake B. M., Salakhutdinov R., Tenenbaum J. B.* Human-Level Concept Learning through Probabilistic Program Induction // Science, 2015.
291. *Lampert C. H., Blaschko M. B., Hofmann T.* Beyond Sliding Windows: Object Localization by Efficient Subwindow Search // 2008 IEEE Conference on Computer Vision and Pattern Recognition, 2008. — P. 1–8.
292. *Landauer T. ., Dumais S. T.* A Solution to Plato's problem: The Latent Semantic Analysis Theory of Acquisition, Induction, and Representation of Knowledge // Psychological review, 1997, vol. 104, no. 2. — P. 211–240.
293. *Lang K. J., Waibel A. H., Hinton G. E.* A Time-Delay Neural Network Architecture for Isolated Word Recognition // Neural Networks, 1990, vol. 3, no. 1. — P. 23–43.
294. Large Automatic Learning, Rule Extraction, and Generalization / J. Denker et al. // Complex Systems, 1987, vol. 1. — P. 877–922.
295. Large Scale Distributed Deep Networks / J. Dean et al. // Proc. 25th NIPS, USA: Curran Associates Inc., 2012. — P. 1223–1231.
296. *Larochelle H., Hinton G. E.* Learning to Combine Foveal Glimpses with a Third-Order Boltzmann Machine // Advances in Neural Information Processing Systems 23 / Curran Associates, Inc., 2010. — P. 1243–1251.
297. A Latent Semantic Model with Convolutional-Pooling Structure for Information Retrieval / Y. Shen et al. // Proc. 23rd CIKM, New York, NY, USA: ACM, 2014. — P. 101–110.
298. *Lavie A., Sagae K., Jayaraman S.* The Significance of Recall in Automatic Metrics for MT Evaluation // Machine Translation: From Real Users to Research: Proc. 6th AMTA / Ed. by R. E. Frederking, K. B. Taylor, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. — P. 134–143.
299. *Lazebnik Y.* Can a Biologist Fix a Radio? — or, What I Learned while Studying Apoptosis // Biochemistry (Moscow), 2004, vol. 69, no. 12. — P. 1403–1406.
300. *Le Q. V., Jaitly N., Hinton G. E.* A Simple Way to Initialize Recurrent Networks of Rectified Linear Units // arXiv, 2015. http://arxiv.org/abs/1504.00941.

301. *Le Q. V., Mikolov T.* Distributed Representations of Sentences and Documents // arXiv, 2014. http://arxiv.org/abs/1405.4053.
302. Learning Author-Topic Models from Text Corpora / M. Rosen-Zvi et al. // ACM Transactions on Information Systems, 2010, vol. 28, no. 1. — P. 1–38.
303. Learning Convolutional Feature Hierarchies for Visual Recognition / K. Kavukcuoglu et al. // Advances in Neural Information Processing Systems 23 / Curran Associates, Inc., 2010. — P. 1090–1098.
304. Learning Deep Structured Semantic Models for Web Search using Clickthrough Data / P.-S. Huang et al., Proc. CIKM, 2013.
305. Learning Longer Memory in Recurrent Neural Networks / T. Mikolov et al. // arXiv, 2014. http://arxiv.org/abs/1412.7753.
306. Learning Mid-Level Features for Recognition / Y. Boureau et al. // Proc. International Conference on Computer Vision and Pattern Recognition (CVPR'10), IEEE, 2010.
307. Learning New Facts from Knowledge Bases with Neural Tensor Networks and Semantic Word Vectors / D. Chen et al. // International Conference on Learning Representations (ICLR), 2013.
308. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation / K. Cho et al. // Proc. 2014 EMNLP, Doha, Qatar: ACL, 2014. — P. 1724–1734.
309. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation / K. Cho et al. // Proc. EMNLP 2014, 2014. — P. 1724–1734.
310. Learning to Navigate in Complex Environments / P. Mirowski et al. // arXiv, 2016. http://arxiv.org/abs/1611.03673.
311. Learning What and Where to Draw / S. E. Reed et al. // arXiv, 2016. http://arxiv.org/abs/1610.02454.
312. Learning where to Attend with Deep Architectures for Image Tracking / M. Denil et al. // arXiv, 2011. http://arxiv.org/abs/1109.3737.
313. *Lebret R., Collobert R.* Word Embeddings through Hellinger PCA // Proc. 14th EACL, Gothenburg, Sweden: ACL, April 2014. — P. 482–490.
314. *LeCun Y., Bengio Y., Hinton G.* Deep learning // Nature, 2015, vol. 521, no. 7553. — P. 436–444.
315. *LeCun Y., Cortes C.* MNIST Handwritten Digit Database, http://yann.lecun.com/exdb/mnist/, 2010.
316. *Lee A. X., Levine S., Abbeel P.* Learning Visual Servoing with Deep Features and Fitted Q-Iteration // arXiv, 2017. http://arxiv.org/abs/1703.11000.
317. *Lee D.-H.* Pseudo-Label : The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks - pseudo_label_final.pdf.
318. *Lei T., Zhang Y.* Training RNNs as Fast as CNNs // arXiv, 2017. http://arxiv.org/abs/1709.02755.
319. *Lennie P.* The Cost of Cortical Computation // Current Biology, 2003, vol. 13, no. 6. — P. 493 – 497.
320. *Lerer A., Gross S., Fergus R.* Learning Physical Intuition of Block Towers by Example // arXiv, 2016. http://arxiv.org/abs/1603.01312.
321. *Levine S., Koltun V.* Guided Policy Search // Proc. 30th ICML / vol. III of *ICML'13*, JMLR.org, 2013. — P. 1–9.
322. *Levy O., Goldberg Y.* Dependency-Based Word Embeddings // Proc. 52nd ACL, Vol. 2: Short Papers, 2014. — P. 302–308.
323. *Levy O., Goldberg Y.* Neural Word Embedding as Implicit Matrix Factorization // Advances in Neural Information Processing Systems 27 / Curran Associates, Inc., 2014. — P. 2177–2185.

324. *Li Y.* Deep Reinforcement Learning: An Overview // arXiv, 2017. http://arxiv.org/abs/1701.07274.
325. *Liang M., Hu X.* Recurrent Convolutional Neural Network for Object Recognition // 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2015. — P. 3367–3375.
326. *Lin C.-Y., Och F. J.* Automatic Evaluation of Machine Translation Quality Using Longest Common Subsequence and Skip-bigram Statistics // Proc. 42nd ACL, Stroudsburg, PA, USA: ACL, 2004.
327. *Lin J.* Divergence Measures Based on the Shannon Entropy // IEEE Transactions on Information Theory, Jan 1991, vol. 37, no. 1. — P. 145–151.
328. *Lin M., Chen Q., Yan S.* Network In Network // arXiv, 2013. http://arxiv.org/abs/1312.4400.
329. *Linnainmaa S.*, The Representation of the Cumulative Rounding Error of an Algorithm as A Taylor Expansion of the Local Rounding Errors, Master's thesis, Univ. Helsinki, 1970.
330. *Liu D. C., Nocedal J.* On the Limited Memory BFGS Method for Large Scale Optimization // Mathematical Programming, 1989, vol. 45, no. 1. — P. 503–528.
331. *Livnat A., Papadimitriou C.* Sex As an Algorithm: The Theory of Evolution Under the Lens of Computation // Communications of the ACM, 2016, vol. 59, no. 11. — P. 84–93.
332. *Lomo T.* Long-Lasting Potentiation of Synaptic Transmission in the Dentate Area of the Anaesthetized Rabbit Following Stimulation of the Perforant Path // Philosophical Transactions of the Royal Society B: Biological Sciences, 2003, vol. 358, no. 1432. — P. 617–620.
333. *Lotter W., Kreiman G., Cox D.* Unsupervised Learning of Visual Structure using Predictive Generative Networks // arXiv, 2015. http://arxiv.org/abs/1511.06380.
334. *Lowe D. G.* Object Recognition from Local Scale-Invariant Features // Proc. 7th IEEE International Conference on Computer Vision / vol. 2, 1999. — P. 1150–1157 vol.2.
335. *Lowe D. G.* Distinctive Image Features from Scale-Invariant Keypoints // International Journal of Computer Vision, 2004, vol. 60, no. 2. — P. 91–110.
336. LSTM: A Search Space Odyssey / K. Greff et al. // arXiv, 2015. http://arxiv.org/abs/1503.04069.
337. LSUN: Construction of a Large-Scale Image Dataset using Deep Learning with Humans in the Loop / F. Yu et al. // arXiv, 2015. http://arxiv.org/abs/1506.03365.
338. *Luo Q., Xu W.* Learning Word Vectors Efficiently Using Shared Representations and Document Representations // Proc. Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI Press, 2015. — P. 4180–4181.
339. *Luo Q., Xu W., Guo J.* A Study on the CBOW Model's Overfitting and Stability // Proc. 5th International Workshop on Web-Scale Knowledge Representation, New York, NY, USA: ACM, 2014. — P. 9–12.
340. *Luong M.-T., Socher R., Manning C. D.* Better Word Representations with Recursive Neural Networks for Morphology // CoNLL, Sofia, Bulgaria: 2013.
341. *Luong T., Pham H., Manning C. D.* Effective Approaches to Attention-based Neural Machine Translation // Proc. 2015 EMNLP, Lisbon, Portugal: ACL, September 2015. — P. 1412–1421.
342. *Maas A. L., Hannun A. Y., Ng A. Y.* Rectifier Nonlinearities Improve Neural Network Acoustic Models // Proc. 29th ICML, 2013, vol. 30, no. 1.
343. *MacKay D. J.* Information Theory, Inference and Learning Algorithms, Cambridge University Press, 2003.
344. *MacKay D. J. C.* A Practical Bayesian Framework for Backpropagation Networks // Neural Computation, 1992, vol. 4, no. 3. — P. 448–472.
345. *Makhzani A., Frey B. J.* PixelGAN Autoencoders // arXiv, 2017. http://arxiv.org/abs/1706.00531.

346. Making Deep Belief Networks Effective for Large Vocabulary Continuous Speech Recognition / T. N. Sainath et al. // ASRU, IEEE, 2011. — P. 30–35.
347. Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings / T. Bolukbasi et al. // arXiv, 2016. http://arxiv.org/abs/1607.06520.
348. *Marcelja S.* Mathematical Description of the Responses of Simple Cortical Cells // Journal of the Optical Society of America, 1980, vol. 70, no. 11. — P. 1297–1300.
349. *Marie B., Max A.* Multi-Pass Decoding With Complex Feature Guidance for Statistical Machine Translation // Proc. 53rd ACL and the 7th IJCNLP, Vol. 2: Short Papers, Beijing, China: ACL, July 2015. — P. 554–559.
350. Markov Topic Models / C. Wang et al. // Journal of Machine Learning Research, 2009, vol. 5. — P. 583–590.
351. *Markram H., Gerstner W., Sjostrom P. J.* A History of Spike-Timing-Dependent Plasticity // Frontiers in Synaptic Neuroscience, 2011, vol. 3, no. 4.
352. *Markram H., Gerstner W., Sjostrom P. J.* Spike Timing–Dependent Plasticity: A Hebbian Learning Rule // Front Synaptic Neuroscience, 2011, vol. 3, no. 4.
353. Mask R-CNN / K. He et al. // arXiv, 2017. http://arxiv.org/abs/1703.06870.
354. Massively Parallel Methods for Deep Reinforcement Learning / A. Nair et al. // arXiv, 2015. http://arxiv.org/abs/1507.04296.
355. Mastering the Game of Go with Deep Neural Networks and Tree Search / D. Silver et al. // Nature, 2016, vol. 529, no. 7587. — P. 484–489.
356. *McCulloch W., Pitts W.* A Logical Calculus of the Ideas Immanent in Nervous Activity // Bulletin of Mathematical Biophysics, 1943, vol. 7. — P. 115–133.
357. *Mescheder L., Nowozin S., Geiger A.* Adversarial Variational Bayes: Unifying Variational Autoencoders and Generative Adversarial Networks // Proc. 34th ICML / vol. 70 of *Proceedings of Machine Learning Research*, PMLR, 2017. — P. 2391–2400.
358. *Metzen J. H.* Variational Autoencoder in TensorFlow, 2015. https://jmetzen.github.io/2015-11-27/vae.html.
359. *Michie D., Chambers R. A.* BOXES: An Experiment in Adaptive Control // Machine Intelligence / Edinburgh, UK: Oliver and Boyd, 1968.
360. *Mikolov T.* Statistical Language Models Based on Neural Networks: Ph.D. thesis / Ph. D. thesis, Brno University of Technology, 2012.
361. *Mikolov T., Joulin A., Baroni M.* A Roadmap towards Machine Intelligence // arXiv, 2015. http://arxiv.org/abs/1511.08130.
362. *Minsky M.* Neural Nets and the Brain Model Problem: Ph.D. thesis / Princeton University, 1954.
363. *Minsky M.* Steps Toward Artificial Intelligence // Computers and Thought / McGraw-Hill, New York, 1963. — P. 406–450.
364. *Minsky M.* Matter, Mind and Models // IFIP Congress, Spartan Books, 1965. — P. 45–50. http://dspace.mit.edu/handle/1721.1/6119.
365. *Minsky M., Papert S.* Perceptrons, Cambridge, MA: MIT Press, 1969.
366. *Mirza M., Osindero S.* Conditional Generative Adversarial Nets // arXiv, 2014. http://arxiv.org/abs/1411.1784.
367. *Mitchell J., Lapata M.* Composition in Distributional Models of Semantics // Cognitive Science, 2010, vol. 34, no. 8. — P. 1388–1429.
368. *Mitchell T. M.* Machine Learning, 1 edition, New York, NY, USA: McGraw-Hill, Inc., 1997.
369. *Mnih A., Hinton G. E.* A Scalable Hierarchical Distributed Language Model // Advances in neural information processing systems, 2009. — P. 1081–1088.
370. *Mnih A., Kavukcuoglu K.* Learning Word Embeddings Efficiently with Noise-Contrastive Estimation // Advances in Neural Information Processing Systems 26 / Curran Associates, Inc., 2013. — P. 2265–2273.

371. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications / A. G. Howard et al. // arXiv, 2017. http://arxiv.org/abs/1704.04861.
372. Modeling Interestingness with Deep Neural Networks / J. Gao et al. // EMNLP, 2014.
373. Modeling User Activities on the Web Using Paragraph Vector / Y. Tagami et al. // Proc. 24th WWW, New York, NY, USA: ACM, 2015. — P. 125–126.
374. *Molchanov D., Ashukha A., Vetrov D.* Variational Dropout Sparsifies Deep Neural Networks // Proc. 34th ICML / vol. 70 of *Proceedings of Machine Learning Research*, PMLR, 2017. — P. 2498–2507.
375. *Moll H., Tomasello M.* Infant cognition // Current Biology, 2010, vol. 20, no. 20. — P. R872–R875.
376. *Mordvintsev A., Olah C., Tyka M.* Inceptionism : Going Deeper into Neural Networks, 2015. http://googleresearch.blogspot.com/2015/06/inceptionism-going-deeper-into-neural.html.
377. *Mozer M. C.* A Focused Backpropagation Algorithm for Temporal Pattern Recognition // Complex Systems, 1989, vol. 3. — P. 349–381.
378. *Mozer M. C.* Neural Net Architectures for Temporal Sequence Processing, Addison-Wesley, 1994. — P. 243–264.
379. *Muller M.* Information Retrieval for Music and Motion, Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.
380. *Muller V. C., Bostrom N.* Future Progress in Artificial Intelligence: A Survey of Expert Opinion // Fundamental Issues of Artificial Intelligence / Ed. by V. C. Muller, Cham: Springer International Publishing, 2016. — P. 555–572.
381. *Murphy K. P.* Machine Learning: a Probabilistic Perspective, Cambridge University Press, 2013.
382. *Nair V., Hinton G. E.* Rectified Linear Units Improve Restricted Boltzmann Machines // Proc. 27th ICML, 2010. — P. 807–814.
383. *Neal R. M.* Bayesian Learning for Neural Networks, Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.
384. *Neelakantan A., Roth B., McCallum A.* Compositional Vector Space Models for Knowledge Base Completion // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 156–166.
385. *Neil D., Pfeiffer M., Liu S.* Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences // Proc. 30th Conference on Neural Information Processing Systems (NIPS 2016) / 2016.
386. *Nesterov Y.* Introductory Lectures on Convex Optimization: A Basic Course, Springer Publishing Company, Incorporated, 2014.
387. Neural Architectures for Named Entity Recognition / G. Lample et al. // arXiv, 2016. http://arxiv.org/abs/1603.01360.
388. Neural Networks : Tricks of the Trade / Ed. by G. B. Orr, K.-R. Mueller, Springer, 1998, vol. 1524 of *Lecture Notes in Computer Science*.
389. Neural Networks : Tricks of the Trade: 2nd Edition / Ed. by G. Montavon et al., Springer, 2012, vol. 7700 of *Lecture Notes in Computer Science*.
390. Neural Probabilistic Language Models / Y. Bengio et al. // Innovations in Machine Learning / Springer, 2006. — P. 137–186.
391. A Neural Probabilistic Structured-Prediction Model for Transition-Based Dependency Parsing / H. Zhou et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 1213–1222.
392. *Newell A., Shaw J. C., Simon H. A.* Report on a General Problem-Solving Program // Proc. International Conference on Information Processing, 1959. — P. 256–264.
393. *Nilsson D.-E., Pelger S.* A Pessimistic Estimate of the Time Required for an Eye to Evolve // Proc. Royal Society of London B: Biological Sciences, 1994, vol. 256, no. 1345. — P. 53–58.

394. Nonparametric Variational Auto-encoders for Hierarchical Representation Learning / P. Goyal et al. // arXiv, 2017. http://arxiv.org/abs/1703.07027.
395. *Nossenson N., Messer H.* Modeling Neuron Firing Pattern Using a Two State Markov Chain // 2010 IEEE Sensor Array and Multichannel Signal Processing Workshop, 2010. — P. 41–44.
396. A Novel Connectionist System for Unconstrained Handwriting Recognition / A. Graves et al. // IEEE Transactions on Pattern Analysis and Machine Intelligence, 2009, vol. 31, no. 5. — P. 855–868.
397. *Nowozin S., Cseke B., Tomioka R.* f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization // Advances in Neural Information Processing Systems 29 / Curran Associates, Inc., 2016. — P. 271–279.
398. On the Number of Linear Regions of Deep Neural Networks / G. Montufar et al. // Proc. 27th NIPS, Cambridge, MA, USA: MIT Press, 2014. — P. 2924–2932.
399. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches / K. Cho et al. // arXiv, 2014. http://arxiv.org/abs/1409.1259.
400. On Unifying Deep Generative Models / Z. Hu et al. // arXiv, 2017. http://arxiv.org/abs/1706.00550.
401. On Using Very Large Target Vocabulary for Neural Machine Translation / S. Jean et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 1–10.
402. One-Shot Learning of Generative Speech Concepts / B. M. Lake et al. // CogSci, cognitivesciencesociety.org, 2014.
403. One-Year-Old Infants Use Teleological Representations of Actions Productively / G. Csibra et al. // Cognitive Science, 2003, vol. 27, no. 1. — P. 111 – 133.
404. OpenWorm: an Open-Science Approach to Modeling Caenorhabditis Elegans / B. Szigeti et al. // Frontiers of Computational Neuroscience, 2014, vol. 2014.
405. *Opper M., Archambeau C.* The Variational Gaussian Approximation Revisited // Neural Computation, 2009, vol. 21, no. 3. — P. 786–792.
406. Overcoming the Curse of Sentence Length for Neural Machine Translation using Automatic Segmentation / J. Pouget-Abadie et al. // arXiv, 2014. http://arxiv.org/abs/1409.1257.
407. OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks / P. Sermanet et al. // arXiv, 2013. http://arxiv.org/abs/1312.6229.
408. *Pantel P.* Inducing Ontological Co-occurrence Vectors // Proc. 43rd ACL, Stroudsburg, PA, USA: ACL, 2005. — P. 125–132.
409. *Paperno D., Pham N. T., Baroni M.* A Practical and Linguistically-Motivated Approach to Compositional Distributional Semantics // Proc. 52nd ACL, Vol. 1: Long Papers, Baltimore, Maryland: ACL, 2014. — P. 90–99.
410. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations / Ed. by D. E. Rumelhart et al., Cambridge, MA, USA: MIT Press, 1986.
411. Parallel Distributed Processing Model with Local Space-Invariant Interconnections and Its Optical Architecture / W. Zhang et al. // Applied Optics, 1990, vol. 29, no. 32.
412. *Pascanu R., Mikolov T., Bengio Y.* On the Difficulty of Training Recurrent Neural Networks // Proc. 30th ICML, 2013. — P. 1310–1318.
413. Path Integral Guided Policy Search / Y. Chebotar et al. // arXiv, 2016. http://arxiv.org/abs/1610.00529.
414. PathNet: Evolution Channels Gradient Descent in Super Neural Networks / C. Fernando et al. // arXiv, 2017. http://arxiv.org/abs/1701.08734.
415. *Pavel M. S., Schulz H., Behnke S.* Recurrent Convolutional Neural Networks for Object-Class Segmentation of RGB-D Video // IJCNN, IEEE, 2015. — P. 1–8.
416. *Pearl J.* Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, NY etc.: Morgan Kaufmann, 1994.

417. *Pennington J., Socher R., Manning C.* Glove: Global Vectors for Word Representation // Proc. 2014 EMNLP, Doha, Qatar: ACL, 2014. — P. 1532–1543.
418. Perception of the Speech Code / A. M. Liberman et al. // Psychology Review, 1967, vol. 74, no. 6. — P. 431–461.
419. *Peters J., Schaal S.* Reinforcement Learning of Motor Skills with Policy Gradients // Neural Networks, 2008, vol. 21, no. 4. — P. 682–697.
420. Phoneme Recognition Using Time-Delay Neural Networks / A. Waibel et al. // IEEE Transactions on Acoustics, Speech, and Signal Processing, Mar 1989, vol. 37, no. 3. — P. 328–339.
421. Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network / C. Ledig et al. // arXiv, 2016. http://arxiv.org/abs/1609.04802.
422. Photovoltaic Retinal Prosthesis: Implant Fabrication and Performance / L. Wang et al. // Journal of Neural Engineering, 2012, vol. 9, no. 4. — P. 046014.
423. *Pilgrim M.* Dive Into Python 3, Berkeley, CA, USA: Apress, 2009.
424. *Pinheiro P. H. O., Collobert R.* Recurrent Convolutional Neural Networks for Scene Parsing // arXiv, 2013. http://arxiv.org/abs/1306.2795.
425. *Pinheiro P. H. O., Collobert R.* Recurrent Convolutional Neural Networks for Scene Labeling // Proc. 31th ICML, 2014. — P. 82–90.
426. Playing Atari With Deep Reinforcement Learning / V. Mnih et al. // NIPS Deep Learning Workshop / 2013.
427. Policy Gradient Methods for Reinforcement Learning with Function Approximation / R. S. Sutton et al. // Proc. 12th NIPS, Cambridge, MA, USA: MIT Press, 1999. — P. 1057–1063.
428. *Pollack J. B., Blair A. D.* Why Did TD-Gammon Work? // NIPS, MIT Press, 1996. — P. 10–16.
429. *Potapenko A., Vorontsov K.* Robust pLSA Performs Better Than LDA // Proc. 35th ECIR / vol. 7814 of *Lecture Notes in Computer Science*, Springer, 2013. — P. 784–787.
430. *Prokhorov D.* A Convolutional Learning System for Object Classification in 3-D LIDAR Data // IEEE Transactions on Neural Networks, 2010, vol. 21, no. 5. — P. 858–863.
431. *McCarthy J., Minsky M. et al.* A proposal for the Dartmouth summer research project on artificial intelligence, 1955. http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html.
432. *Radford A., Metz L., Chintala S.* Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks // arXiv, 2015. http://arxiv.org/abs/1511.06434.
433. *Raiko T., Valpola H., LeCun Y.* Deep Learning Made Easier by Linear Transformations in Perceptrons. // AISTATS / vol. 22 of *JMLR Proceedings*, JMLR.org, 2012. — P. 924–932.
434. *Raina R., Madhavan A., Ng A.* Large-Scale Deep Unsupervised Learning Using Graphics Processors // Proc. 26th ICML, 2009. — P. 873–880.
435. *Rardin R. L.* Optimization in Operations Research, Pearson, 2016.
436. RC-NET: A General Framework for Incorporating Knowledge into Word Representations / C. Xu et al. // Proc. 23rd ACM International Conference on Conference on Information and Knowledge Management, New York, NY, USA: ACM, 2014. — P. 1219–1228.
437. Reasoning With Neural Tensor Networks for Knowledge Base Completion / R. Socher et al. // Advances in Neural Information Processing Systems (NIPS), 2013.
438. Recent Advances in Convolutional Neural Networks / J. Gu et al. // arXiv, 2015. http://arxiv.org/abs/1512.07108.
439. Recurrent Batch Normalization / T. Cooijmans et al. // arXiv, 2016. http://arxiv.org/abs/1603.09025.
440. Recurrent Highway Networks / J. G. Zilly et al. // Proc. 34th ICML / vol. 70 of *Proceedings of Machine Learning Research*, PMLR, 2017. — P. 4189–4198.
441. Recurrent Models of Visual Attention / V. Mnih et al. // Advances in Neural Information Processing Systems 27 / Curran Associates, Inc., 2014. — P. 2204–2212.

442. Recurrent Neural Network Based Language Model / T. Mikolov et al. // INTERSPEECH, 2010, vol. 2. — P. 3.
443. Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank / R. Socher et al. // Proc. EMNLP 2013 / vol. 1631, 2013. — P. 1642.
444. *Redmon J., Farhadi A.* YOLO9000: Better, Faster, Stronger // arXiv, 2016. http://arxiv.org/abs/1612.08242.
445. *Rehurek R.* A Path to Unsupervised Learning through Adversarial Networks, 2013. https://rare-technologies.com/deep-learning-with-word2vec-and-gensim/.
446. *Rehurek R., Sojka P.* Software Framework for Topic Modelling with Large Corpora // Proc. LREC 2010 Workshop on New Challenges for NLP Frameworks, Valletta, Malta: ELRA, 2010. — pp. 45–50.
447. *Rensink R. A.* The Dynamic Representation of Scenes // Visual Cognition, 2000, vol. 7, no. 1–3. — P. 17–42.
448. Rethinking the Inception Architecture for Computer Vision / C. Szegedy et al. // arXiv, 2015. http://arxiv.org/abs/1512.00567.
449. Retrofitting Word Vectors to Semantic Lexicons / M. Faruqui et al. // Proc. 2015 NAACL-HLT, Denver, Colorado: ACL, May–June 2015. — P. 1606–1615.
450. Robust Adversarial Reinforcement Learning / L. Pinto et al. // Proc. 34th ICML / vol. 70 of *Proceedings of Machine Learning Research*, PMLR, 2017. — P. 2817–2826.
451. *Rohas R.* Neural Networks — A Systematic Introduction, Springer-Verlag, Berlin, New-York, 1996.
452. *Rohde D. L. T., Gonnerman L. M., Plaut D. C.* An Improved Model of Semantic Similarity Based on Lexical Co-Occurence // Communications of the ACM, 2006, vol. 8. — P. 627–633.
453. *Rong X.* word2vec Parameter Learning Explained // arXiv, 2014. http://arxiv.org/abs/1411.2738.
454. *Ronneberger O., Fischer P., Brox T.* U-Net: Convolutional Networks for Biomedical Image Segmentation // arXiv, 2015. http://arxiv.org/abs/1505.04597.
455. *Rosenblatt F.* The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain // Psychological Review, 1958, vol. 65, no. 6. — P. 386–408.
456. *Rosenblatt F.* Principles of Neurodynamics, Spartan, New York, 1962.
457. *Rubenstein H., Goodenough J. B.* Contextual Correlates of Synonymy // Communications of the ACM, 1965, vol. 8, no. 10. — P. 627–633.
458. *Ruder S.* An Overview of Gradient Descent Optimization Algorithms // arXiv, 2016. http://arxiv.org/abs/1609.04747.
459. *Rumelhart D. E., Hinton G. E., Williams R. J.* Learning Internal Representations by Error Propagation // Parallel Distributed Processing. Vol 1: Foundations / Cambridge, MA, USA: MIT Press, 1986.
460. *Rumelhart D. E., Hinton G. E., Williams R. J.* Learning Representations by Back-propagating Errors // Neurocomputing: Foundations of Research / Cambridge, MA, USA: MIT Press, 1988. — P. 696–699.
461. *Saatchi Y., Wilson A. G.* Bayesian GAN // arXiv, 2017. http://arxiv.org/abs/1705.09558.
462. *Sadrzadeh M., Grefenstette E.* A Compositional Distributional Semantics, Two Concrete Constructions, and Some Experimental Evaluations // Proc. 5th International Conference on Quantum Interaction, Berlin, Heidelberg: Springer-Verlag, 2011. — P. 35–47.
463. *Sahlgren M.* The Distributional Hypothesis // Italian Journal of Linguistics, 2008, vol. 20, no. 1. — P. 33–54.
464. *Salakhutdinov R.* Learning Deep Boltzmann Machines using Adaptive MCMC // Proc. 27th ICML, 2010. — P. 943–950.
465. *Salakhutdinov R.* Learning Deep Generative Models // Annual Review of Statistics and Its Application, 2015, vol. 2, no. 1. — P. 361–385.

466. *Salakhutdinov R., Hinton G. E.* Deep Boltzmann Machines // Proc. Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater Beach, Florida, USA, April 16-18, 2009, 2009. — P. 448–455.
467. *Salakhutdinov R., Hinton G. E.* A Better Way to Pretrain Deep Boltzmann Machines // Advances in Neural Information Processing Systems 25, 2012. — P. 2456–2464.
468. *Salakhutdinov R., Larochelle H.* Efficient Learning of Deep Boltzmann Machines // Proc. 13th International Conference on Artificial Intelligence and Statistics (AISTATS), 2010. — P. 693–700.
469. *Sallans B., Hinton G. E.* Reinforcement Learning with Factored States and Actions // Journal of Machine Learning Research, 2004, vol. 5. — P. 1063–1088.
470. *Savage L.* The Foundations of Statistics, New York: Wiley, 1954.
471. *Scherer D., Muller A., Behnke S.* Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition // Proc. 20th International Conference on Artificial Neural Networks: Part III, Berlin, Heidelberg: Springer-Verlag, 2010. — P. 92–101.
472. *Schmidhuber J.* Learning Complex, Extended Sequences Using the Principle of History Compression // Neural Computation, 1992, vol. 4, no. 2. — P. 234–242.
473. *Schmidhuber J.* Low-Complexity Art // Leonardo, 1997, vol. 30, no. 2. — P. 97–103.
474. *Schmidhuber J.* Simple Algorithmic Principles of Discovery, Subjective Beauty, Selective Attention, Curiosity & Creativity // Discovery Science: 10th International Conference, DS 2007 Sendai, Japan, October 1-4, 2007. Proceedings / Ed. by V. Corruble et al., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. — P. 26–38.
475. *Schmidhuber J.* Deep Learning in Neural Networks: An Overview // Neural Networks, 2015, vol. 61. — P. 85–117.
476. *Schwenk H.* Continuous Space Language Models // Comput. Speech Lang., 2007, vol. 21, no. 3. — P. 492–518.
477. *Searle J. R.* Minds, Brains and Programs // Behavioral and Brain Sciences, 1980, vol. 3, no. 3. — P. 417–57.
478. *Sedol L.* Анализ 5-й партии матча с AlphaGo, 23 марта 2016 г., http://news.donga.com/Issue/List/08000000000070/3/08000000000070/20160323/77152796/1, перевод на английский см. на http://badukinkorea.tumblr.com/post/142578829241/lee-sedols-review-on-the-fifth-match-with-alphago.
479. *Sedol L.* Интервью, 29 января 2016 г., http://news.donga.com/3/all/20160129/76197440/1, частичный перевод на русский см. на http://gofederation.ru/board/viewtopic.php?f=2&t=656.
480. *Segee B.* Methods in Neuronal Modeling: from Ions to Networks, 2nd Edition // Computing in Science Engineering, 1999, vol. 1, no. 1. — P. 81–81.
481. *Seide F., Li G., Yu D.* Conversational Speech Transcription Using Context-Dependent Deep Neural Networks // Interspeech 2011, International Speech Communication Association, 2011.
482. Self-Normalizing Neural Networks / G. Klambauer et al. // arXiv, 2017. http://arxiv.org/abs/1706.02515.
483. Semantic Compositionality Through Recursive Matrix-vector Spaces / R. Socher et al. // Proc. 2012 Joint EMNLP and Computational Natural Language Learning, Stroudsburg, PA, USA: ACL, 2012. — P. 1201–1211.
484. Semantic Object Parsing with Graph LSTM / X. Liang et al. // Proc. ECCV 2016, Part I / Ed. by B. Leibe et al., Cham: Springer International Publishing, 2016. — P. 125–143.
485. Semantically Smooth Knowledge Graph Embedding / S. Guo et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 84–94.
486. Semeval-2014 Task 1: Evaluation of Compositional Distributional Semantic Models on Full Sentences through Semantic Relatedness and Textual Entailment / M. Marelli et al. // SemEval-2014, 2014.

487. Semi-supervised Learning with Ladder Networks / A. Rasmus et al. // Proc. 28th NIPS, Cambridge, MA, USA: MIT Press, 2015. — P. 3546–3554.
488. *Sennrich R., Haddow B., Birch A.* Edinburgh Neural Machine Translation Systems for WMT 16 // arXiv, 2016. http://arxiv.org/abs/1606.02891.
489. Sex, Mixability, and Modularity / A. Livnat et al. // Proc. National Academy of Sciences, 2010, vol. 107, no. 4. — P. 1452–1457.
490. *Shannon C.* This Mouse Is Smarter Than You Are // Popular Science, 1952. — P. 99–101.
491. *Shaw Z. A.* Learn Python the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code, 3rd edition, Addison-Wesley Professional, 2013.
492. Shift-Invariant Pattern Recognition Neural Network and Its Optical Architecture / W. Zhang et al. // Proceedings of Annual Conference of the Japan Society of Applied Physics, 1988.
493. *Shimodaira H.* Improving Predictive Inference under Covariate Shift by Weighting the Log-Likelihood Function // Journal of Statistical Planning and Inference, 2000, vol. 90, no. 2. — P. 227–244.
494. *Shortliffe E. H., Buchanan B. G.* A Model of Inexact Reasoning in Medicine // Mathematical Biosciences, 1975, vol. 23. — P. 351–379.
495. Show and Tell: A Neural Image Caption Generator / O. Vinyals et al. // arXiv, 2014. http://arxiv.org/abs/1411.4555.
496. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention / K. Xu et al. // arXiv, 2015, vol. 2, no. 3. — P. 5. http://arxiv.org/abs/1502.03044.
497. *Sjostrom J., Gerstner W.* Spike-Timing Dependent Plasticity // Scholarpedia, 2010, vol. 5, no. 2. — P. 1362.
498. Skip-Thought Vectors / R. Kiros et al. // Advances in Neural Information Processing Systems 28 / Curran Associates, Inc., 2015. — P. 3294–3302.
499. *Smolensky P.* Information Processing in Dynamical Systems: Foundations of Harmony Theory // Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1 / Cambridge, MA, USA: MIT Press, 1986. — P. 194–281.
500. Snapshot Ensembles: Train 1, Get M for Free / G. Huang et al. // arXiv, 2017. http://arxiv.org/abs/1704.00109.
501. *Soricut R., Och F.* Unsupervised Morphology Induction Using Word Embeddings // Proc. 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Denver, Colorado: ACL, 2015. — P. 1627–1637.
502. *Spelke E. S., Gutheil G., Van de Walle G.* The Development of Object Perception // Visual Cognition: An Invitation to Cognitive Science / 1995.
503. *Spelke E. S., Kinzler K. D.* Core Knowledge // Developmental Science, 2007, vol. 10, no. 1. — P. 89–96.
504. SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and <1MB Model Size / F. N. Iandola et al. // arXiv, 2016. http://arxiv.org/abs/1602.07360.
505. *Srivastava N., Salakhutdinov R.* Multimodal Learning with Deep Boltzmann Machines // Journal of Machine Learning Research, 2014, vol. 15, no. 1. — P. 2949–2980.
506. *Srivastava R. K., Greff K., Schmidhuber J.* Highway Networks // arXiv, 2015. http://arxiv.org/abs/1505.00387.
507. *Srivastava R. K., Greff K., Schmidhuber J.* Training Very Deep Networks // Proc. 28th NIPS, Cambridge, MA, USA: MIT Press, 2015. — P. 2377–2385.
508. StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks / H. Zhang et al. // arXiv, 2016. http://arxiv.org/abs/1612.03242.
509. Statistical Machine Translation Features with Multitask Tensor Networks / H. Setiawan et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 31–41.

510. *Stenetorp P.* Transition-based Dependency Parsing Using Recursive Neural Networks // Deep Learning Workshop at NIPS 2013, 2013.
511. Striving for Simplicity: The All Convolutional Net / J. T. Springenberg et al. // arXiv, 2014. http://arxiv.org/abs/1412.6806.
512. Structured Training for Neural Network Transition-Based Parsing / D. Weiss et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 323–333.
513. A Study of Translation Edit Rate with Targeted Human Annotation / M. Snover et al. // Proc. Association for Machine Translation in the Americas, 2006. — P. 223–231.
514. *Sugiyama M., Krauledat M., Muller K.-R.* Covariate Shift Adaptation by Importance Weighted Cross Validation // Journal of Machine Learning Research, 2007, vol. 8. — P. 985–1005.
515. A Survey of Monte Carlo Tree Search Methods / C. B. Browne et al. // IEEE Transactions on Computational Intelligence and AI in Games, March 2012, vol. 4, no. 1. — P. 1–43.
516. *Sutskever I., Tieleman T.* On the Convergence Properties of Contrastive Divergence. // AISTATS / vol. 9 of *JMLR Proceedings*, JMLR.org, 2010. — P. 789–795.
517. *Sutskever I., Vinyals O., Le Q. V.* Sequence to Sequence Learning with Neural Networks // arXiv, 2014. http://arxiv.org/abs/1409.3215.
518. *Sutton C., McCallum A.* An Introduction to Conditional Random Fields for Relational Learning, 2006.
519. *Sutton R. S., Barto A. G.* Reinforcement Learning: An Introduction, Cambridge, MA, 1998.
520. Syntax-based Simultaneous Translation through Prediction of Unseen Syntactic Constituents / Y. Oda et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 198–207.
521. Target-driven Visual Navigation in Indoor Scenes using Deep Reinforcement Learning / Y. Zhu et al. // arXiv, 2016. http://arxiv.org/abs/1609.05143.
522. A Taxonomy of Deep Convolutional Neural Nets for Computer Vision / S. Srinivas et al. // arXiv, 2016. http://arxiv.org/abs/1601.06615.
523. *Abadi M., Agarwal A. et al.* TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015, Software available from tensorflow.org. http://tensorflow.org/.
524. *Tesauro G.* Neurogammon Wins Computer Olympiad // Neural Computation, 1989, vol. 1, no. 3. — P. 321–323.
525. *Tesauro G.* Practical Issues in Temporal Difference Learning // Machine Learning, 1992, vol. 8. — P. 257–277.
526. *Tesauro G.* Temporal Difference Learning and TD-Gammon // Communications of the ACM, 1995, vol. 38, no. 3. — P. 58–68.
527. *Thaler L., Arnott S. R., Goodale M. A.* Human Echolocation I // Journal of Vision, 2010, vol. 10, no. 7. — P. 1050.
528. Theano: a CPU and GPU Math Expression Compiler / J. Bergstra et al. // Proc. Python for Scientific Computing Conference (SciPy), 2010, Oral Presentation.
529. *Bastien F., Lamblin P. et al.* Theano: New Features and Speed Improvements, Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.
530. *Thorndike E. L.* Animal Intelligence: An Experimental Study of the Associative Processes in Animals. Psychological Review Monographs, New York: Macmillan, 1898.
531. *Tiedemann J.* News from OPUS - A Collection of Multilingual Parallel Corpora with Tools and Interfaces // Recent Advances in Natural Language Processing (vol V) / Amsterdam/Philadelphia: John Benjamins, 2009. — P. 237–248.
532. *Tieleman T., Hinton G.* Lecture 6.5 — RMSProp: Divide the Gradient by a Running Average of its Recent Magnitude, Coursera: Neural Networks for Machine Learning, 2012. https://www.coursera.org/learn/neural-networks.

533. TinyFlow: Build Your Own DL System in 2K Lines, https://github.com/tqchen/tinyflow, 2016.
534. *Tishby N., Levin E., Solla S.* Consistent Inference of Probabilities in Layered Networks: Predictions and Generalizations // IJCNN International Joint Conference on Neural Networks / vol. II, IEEE, 1989. — P. 403–409.
535. *Titov I., Henderson J.* A Latent Variable Model for Generative Dependency Parsing // Proc. 10th International Conference on Parsing Technologies, Stroudsburg, PA, USA: ACL, 2007. — P. 144–155.
536. Towards a Virtual C. Elegans: A Framework for Simulation and Visualization of the Neuromuscular System in a 3D Physical Environment / A. Palyanov et al. // In silico Biology, 2012, vol. 11, no. 3, 4. — P. 137–147.
537. *Lowe R., Noseworthy M. et al.* Towards an Automatic Turing Test: Learning to Evaluate Dialogue Responses, Submitted to ICLR 2017, 2017. https://openreview.net/forum?id=HJ5PIaseg.
538. Towards Biologically Plausible Deep Learning / Y. Bengio et al. // arXiv, 2015. http://arxiv.org/abs/1502.04156.
539. Transition-based Dependency Parsing Using Two Heterogeneous Gated Recursive Neural Networks / X. Chen et al. // Proc. EMNLP 2015, Lisbon, Portugal: ACL, 2015. — P. 1879–1889.
540. Transition-Based Dependency Parsing with Stack Long Short-Term Memory / C. Dyer et al. // Proc. 53rd ACL and the 7th IJCNLP, Vol. 1: Long Papers, Beijing, China: ACL, 2015. — P. 334–343.
541. *Tsitsiklis J. N., Van Roy B.* An Analysis of Temporal-Difference Learning with Function Approximation // IEEE Transactions on Automatic Control, 1997, vol. 42, no. 5. — P. 674–690.
542. *Turing A. M.* Intelligent Machinery, A Heretical Theory, Lecture given at Oxford, 1948.
543. *Turing A. M.* Computing Machinery and Intelligence, 1950, One of the most influential papers in the history of the cognitive sciences: http://cogsci.umn.edu/millennium/final.html.
544. *Ungerleider L. G., Mishkin M.* Two Cortical Visual Systems // Analysis of Visual Behaviour / Ed. by D. J. Ingle et al., 1982. — P. 549–586.
545. Unrolled Generative Adversarial Networks / L. Metz et al. // arXiv, 2016. http://arxiv.org/abs/1611.02163.
546. An Updated Set of Basic Linear Algebra Subprograms (BLAS) / L. Blackford et al. // ACM Transactions on Mathematical Software, 2002, vol. 28, no. 2. — P. 135–151.
547. *van den Oord A., Dieleman S., Zen H.* WaveNet: A Generative Model for Raw Audio, 2016. https://deepmind.com/blog/wavenet-generative-model-raw-audio/.
548. *van den Oord A., Kalchbrenner N., Kavukcuoglu K.* Pixel Recurrent Neural Networks // arXiv, 2016. http://arxiv.org/abs/1601.06759.
549. Variational Approaches for Auto-Encoding Generative Adversarial Networks / M. Rosca et al. // arXiv, 2017. http://arxiv.org/abs/1706.04987.
550. Variational Lossy Autoencoder / X. Chen et al. // arXiv, 2016. http://arxiv.org/abs/1611.02731.
551. *Vinyals O., Le Q. V.* A Neural Conversational Model // ICML Deep Learning Workshop, arXiv:1506.05869, 2015.
552. *Viola P. A., Jones M. J.* Rapid Object Detection using a Boosted Cascade of Simple Features // Proc. CVPR 2001, 2001. — P. 511–518.
553. *Vorontsov K.* Additive Regularization for Topic Models of Text Collections // Doklady Mathematics, 2014, vol. 89, no. 3. — P. 301–304.
554. *vos Savant M.* Ask Marilyn // Parade, 1990, no. 16,25.
555. The Wake-Sleep Algorithm for Unsupervised Neural Networks / G. E. Hinton et al. // Science, 1995, vol. 268. — P. 1158–1161.

556. *Wald A.* Statistical Decision Functions // The Annals of Mathematical Statistics, 1949, vol. 20, no. 2. — P. 165–205.
557. *Wang C., Blei D. M., Heckerman D.* Continuous Time Dynamic Topic Models // Proc. 24th Conference on Uncertainty in Artificial Intelligence, 2008.
558. *Wang X., McCallum A.* Topics over Time: a Non-Markov Continuous-Time Model of Topical Trends // Proc. 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA: ACM, 2006. — P. 424–433.
559. *Watkins C. J. C. H., Dayan P.* Q-learning // Machine Learning, 1992. — P. 279–292.
560. WaveNet: A Generative Model for Raw Audio / A. van den Oord et al. // arXiv, 2016. http://arxiv.org/abs/1609.03499.
561. Weakly Supervised Memory Networks / S. Sukhbaatar et al. // arXiv, 2015. http://arxiv.org/abs/1503.08895.
562. Webvision: The Organization of the Retina and Visual System, 2007. http://webvision.med.utah.edu/book/.
563. *Weizenbaum J.* ELIZA — A Computer Program for the Study of Natural Language Communication Between Man and Machine // Communications of the ACM, 1966, vol. 9, no. 1. — P. 36–45.
564. *Werbos P. J.* Applications of Advances in Nonlinear Sensitivity Analysis // Proc. 10th IFIP Conference, 31.8 - 4.9, NYC, 1981. — P. 762–770.
565. *Weston J., Chopra S., Bordes A.* Memory networks // arXiv, 2014. http://arxiv.org/abs/1410.3916.
566. *Weston J., Ratle F., Collobert R.* Deep Learning via Semi-supervised Embedding // Proc. 25th ICML, New York, NY, USA: ACM, 2008. — P. 1168–1175.
567. Why Does Unsupervised Pre-Training Help Deep Learning? / D. Erhan et al. // Journal of Machine Learning Research, 2010, vol. 11. — P. 625–660.
568. *Wiener N.* Cybernetics, Second Edition: Or the Control and Communication in the Animal and the Machine, The MIT Press, 1965.
569. *Wiesel D. H., Hubel T. N.* Receptive Fields of Single Neurones in the Cat's Striate Cortex // Journal of Physiology, 1959, vol. 148. — P. 574–591.
570. *Wiesler S., Ney H.* A Convergence Analysis of Log-Linear Training. // NIPS, 2011. — P. 657–665.
571. *Williams R. J.* Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning // Machine Learning, 1992, vol. 8, no. 3-4. — P. 229–256.
572. *Wohler C., Anlauf J. K.* An Adaptable Time-Delay Neural-Network Algorithm for Image Sequence Analysis // IEEE Transactions on Neural Networks, Nov 1999, vol. 10, no. 6. — P. 1531–1536.
573. *Xu R., Wunsch D.* Clustering, Wiley-IEEE Press, 2008.
574. *Yang X., Kwitt R., Niethammer M.* Fast Predictive Image Registration // arXiv, 2016. http://arxiv.org/abs/1607.02504.
575. *Yao K., Zweig G., Peng B.* Attention with Intention for a Neural Network Conversation Model // arXiv, 2015. http://arxiv.org/abs/1510.08565.
576. You Only Look Once: Unified, Real-Time Object Detection / J. Redmon et al. // arXiv, 2015. http://arxiv.org/abs/1506.02640.
577. *Yu M., Dredze M.* Improving Lexical Embeddings with Semantic Knowledge // Proc. 52nd ACL, Vol. 2, 2014. — P. 545–550.
578. *Yudkowsky E.* Artificial Intelligence as a Positive and Negative Factor in Global Risk // Global Catastrophic Risks, 2008, vol. 1. — P. 303.
579. *Yudkowsky E.* Friendly Artificial Intelligence // Singularity Hypotheses: A Scientific and Philosophical Assessment / Berlin: Springer, 2012.
580. *Yudkowsky E.* Harry Potter and the Methods of Rationality, http://www.hpmor.com/, 2015.

581. *Yudkowsky E.* Rationality: From AI to Zombies, Berkeley, CA: Machine Intelligence Research Institute, 2015. http://intelligence.org/rationality-ai-zombies/.
582. *Zaremba W., Sutskever I.* Reinforcement Learning Neural Turing Machines // arXiv, 2015. http://arxiv.org/abs/1505.00521.
583. *Zaremba W., Sutskever I., Vinyals O.* Recurrent Neural Network Regularization // arXiv, 2014. http://arxiv.org/abs/1409.2329.
584. *Zeiler M. D.* ADADELTA: An Adaptive Learning Rate Method // arXiv, 2012. http://arxiv.org/abs/1212.5701.
585. *Zeiler M. D., Fergus R.* Visualizing and Understanding Convolutional Networks // arXiv, 2013. http://arxiv.org/abs/1311.2901.
586. *Zhang X., LeCun Y.* Text Understanding from Scratch // arXiv, 2015. http://arxiv.org/abs/1502.01710.
587. *Zhang X., Zhao J., LeCun Y.* Character-level Convolutional Networks for Text Classification // Advances in Neural Information Processing Systems 28 / Curran Associates, Inc., 2015. — P. 649–657.
588. *Ивахненко А. Г., Лапа В. Г.* Кибернетические предсказывающие устройства. Киев, Наук. думка, 1965.
589. *Колмогоров А. Н.* О представлении непрерывных функций нескольких переменных в виде суперпозиций непрерывных функций одной переменной и сложения // Доклады Академии наук. 1957. Т. 114, № 5. — С. 953–956.
590. *Колмогоров А. Н.* Автоматы и жизнь // Возможное и невозможное в кибернетике / 1964. — С. 10–29.
591. *Лурия А. Р.* Мозг и психические процессы. В 2-х тт. М., Издательство АПН РСФСР, 1970.
592. *Лурия А. Р.* Основы нейропсихологии. М., 1973.
593. *Малых В.* Обучение с подкреплением: от Павлова до игровых автоматов. 2017. https://habrahabr.ru/post/322404/.
594. *Тулупьев А. Л., Николенко С. И., Сироткин А. В.* Байесовские сети: логико-вероятностный подход. СПб.: Наука, 2006. 608 с.
595. *Феллер В.* Введение в теорию вероятностей и её приложения. М., Мир, 1984.
596. *Ширяев А.* Вероятность. М., Наука, 1989.
597. *Ширяев А., Эрлих И., Яськов П.* Вероятность в теоремах и задачах (с доказательствами и решениями). М., МЦНМО, 2013.

С. Николенко, А. Кадурин, Е. Архангельская

Глубокое обучение

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Е. Рафалюк-Бузовская*
Художественный редактор	*С. Заматевская*
Корректоры	*О. Андриевич, О. Порохнявая*

Изготовлено в России. Изготовитель: ООО «Питер Пресс».
Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 11.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 25.10.17. Формат 70×100/16. Усл. п. л. 38,700. Тираж 1200. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87